Palo Alto Research Center

Formalizing the Analysis of Algorithms

By Lyle Harold Ramshaw

XEROX

Formalizing the Analysis of Algorithms

by Lyle Harold Ramshaw

CSL-79-5 June 1979

Abstract: See page iii.

This report reproduces a dissertation submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy. It is also available as Stanford University Computer Science Department technical report STAN-CS-79-741.

This research and printing was supported in part by National Science Foundation grant MCS-77-23738, by Office of Naval Research contract N00014-76-C-0330, by IBM Corporation, and by Xerox Corporation. Reproduction in whole or in part is permitted for any purpose of the United States government.

CR Categories: 5.21, 5.24, 5.25.

Key words and phrases: analysis of algorithms, formal systems, measure theory, probabilistic semantics, program verification, recurrence relations.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Abstract

Consider the average case analyses of particular deterministic algorithms. Typical arguments in this area can be divided into two phases. First, by using knowledge about what it means to execute a program, an analyst characterizes the probability distribution of the performance parameter of interest by means of some mathematical construct, often a recurrence relation. In the second phase, the solution of this recurrence is studied by purely mathematical techniques. Our goal is to build a formal system in which the first phases of these arguments can be reduced to symbol manipulation.

Formal systems currently exist in which one can reason about the correctness of programs by manipulating predicates that describe the state of the executing process. The construction and use of such systems belongs to the field of program verification. We want to extend the ideas of program verification, in particular, the partial correctness techniques of Floyd and Hoare, to allow assertions that describe the probabilisitic state of the executing process to be written and manipulated. Ben Wegbreit proposed a system that extended Floyd-Hoare techniques to handle performance analyses, and we shall take Wegbreit's system as our starting point. Our efforts at formal system construction will also lead us to a framework for program semantics in which programs are interpreted as linear functions between vector spaces of measures. This framework was recently developed by Dexter Kozen, and we shall draw upon his results as well.

We shall call our formal system the *frequency system*. The atomic assertions in this system specify the frequencies with which Floyd-Hoare predicates hold. These atomic assertions are combined with logical and arithmetic connectives to build assertions, and the rules of the frequency system describe how these assertions change as the result of executing program statements. The rules of the frequency system are sound, but not complete.

We then discuss the use of the frequency system in several average case analyses. In our examples, symbol manipulation in the frequency system leads directly to the recurrence relation that describes the distribution of the chosen performance parameter. The last of these examples is the algorithm that performs a straight insertion sort.

Preface

There is one nontechnical problem in the area of formalizing the analysis of algorithms that deserves some discussion: a problem of nomenclature. Instead of dealing with probability, it turns out to be better to deal with a quantity that is the same as probability in every way except that it does not necessarily sum to unity. I chose to call this quantity *frequency*, and to write it "Fr" by analogy with the "Pr" notation for probability. But now the problem: What word is to "frequency" as the word "probabilistic" is to "probability"? For example, if I am thinking of the state of a process as characterized by the probabilities of various events, I am considering a *probabilistic state*; I can describe that state by *probabilistic assertions.* If frequencies instead of probabilities are underneath it all, what should the corresponding terms be?

By quizzing my friends and associates, I came up with four possible solutions to this problem.

(i) Invent the new word "frequentistic".

(ii) Invent the new word "frequencistic".

(iii) Use the word "frequency" as if it were an adjective.

(iv) Use a hyphenated term such as "frequency-based".

The amount of controversy that surrounded my survey indicates that none of these solutions is completely satisfactory. Of the new words, most people seemed to think that "frequentistic" was more euphonious, but that "frequencistic" was a more logical choice. Option (iii) has the difficulty that using the terms *frequency state* and *frequency assertion* seems to demand that we also adopt the terms *probability state* and *probability assertion* instead of those given above, in order to preserve the frequency-probability parallelism. It seems a shame to abandon the perfectly good word "probabilistic" just because it has no exact frequency parallel. In Option (iv), the frequency parallel of the word "probabilistic" is the word "frequency-based"; this hyphenated term is not as exact a parallel as either of the new words, but it has the distinct advantage of being English.

At least one person voted for each of the options, although people generally agreed that they were picking the best of a bad lot. I chose to adopt Option (i). Those readers who find themselves unable to adjust to "frequentistic" might be encouraged by the fact that their taste on this issue agrees with Don Knuth's; Don argued in favor of a combination of Options (iii) and (iv). If the term "frequentistic" catches on, at least this problem will be solved for future authors in the field. If not, they will have to reopen the nomenclatural negotiations.

My use of the term "vanilla" with a technical meaning also raised some storms of protest. I agree that "vanilla" is both nonspecific and undignified, but I heard no better suggestions. In addition, I plead that the concept to which it refers probably demands as much improvement as the term. Basic terminology should receive careful consideration, but one shouldn't waste time searching for the perfect name for a peripheral idea. In a class that I took, Mike Spivak used the single adjective "yellow" for each new concept that he introduced; only after that concept had been developed somewhat would he replace "yellow" with a more suggestive term.

I would like to thank my adviser, Don Knuth, for countless exhortations and consistently excellent guidance. He provides a formidable personal and professional example to his students. It has been a joy to work with someone as dedicated as Don; I have the feeling that he would lose sleep if he thought that he had done something that lowered the position of Computer Science in the academic community. Leo Guibas and Andy Yao, my other readers, earned my gratitude for their technical camaraderie, and especially for their expeditious perusal of my tardy draft.

This thesis was produced with the TEX system for technical text [21]. I thank Don Knuth, who courageously built this system; Leo Guibas, who gallantly transported it to the Xerox Palo Alto Research Center (PARC); PARC itself, where most of the production took place; and Xerox ASD, whose magnificent hardware printed the originals of these pages.

Thanks also go to Dana Scott, for introducing me to Dexter Kozen's work; and to Bob Sedgewick, for coming up with the terms "in-branch" and "out-branch", and for suggesting that I include the nomenclatural discussion above. Finally, I express my miscellaneous gratitude to Al Aho, Sam Bent, Jim Boyce, Mark Brown, Ole-Johan Dahl, John Gilbert, Daniel Greene, David Gries, David Jefferson, Bill Laaser, Brian McCune, Greg Nelson, Terry Roberts, and Bill van Melle.

Table of Contents

Chapter 1. Introduction

The Analysis of Algorithms.

The analysis of algorithms, or "algorithmic analysis" for short, is mathematical reasoning about the properties of algorithms that solve a particular abstract problem [19]. Algorithmic analysis strives for precise theoretical understandings; hence, there is usually no hope of significant progress when the problem under consideration is as large and as arbitrary as is common in the real world. Instead, the algorithmic analyst concentrates on the cleaner problems that can be found in more abstract, well-behaved disciplines. One catalogue of these areas appears as the table of contents of the book *The Design and Analysis of Computer Algorithms* by Aho, Hopcroft, and Ullman [1]. When expertly carried out, this restriction to more tractable domains does not vitiate the relevance of the results. The large and complex problems that the real world presents are usually best solved by adroit combinations of the techniques that arise in simpler contexts. In adition, many large and complex programs spend most of their time executing in the small regions known as *inner loops*; these inner loops are more likely to be analytically tractable.

Suppose that we have a suitably clean problem in mind, such as sorting an array of numbers, computing the transitive closure of a directed graph, or searching for a pattern in a text string. There are two different types of analyses that we can attempt to perform. First, we can take a particular algorithm that solves the problem, and analyze some facet of the performance of that algorithm. We might determine the amount of some computational resource that it demands in the worst case. Or, choosing some probability distribution for the space of inputs, we could attempt the often more difficult task of computing that algorithm's behavior on the average. In analyses of this first type, then, we apply our mathematics to a particular algorithm that solves the abstract problem.

In the second type of analysis, on the other hand, we consider instead a class of algorithms that solve the problem, and attempt to derive information about this class as a whole. The class under consideration usually consists of all algorithms that have a certain form, such as all decision trees, or all programs for a certain specified variety of abstract automaton. Given such a class, we can attempt to discover the properties of the optimal algorithm in the class, the one that is the most efficient in some measure. In analyses of this second kind, we are applying mathematics to the questions generated by the combination of the abstract problem and the particular class of algorithms that we have chosen.

Both of these types of analyses are important. We are going to focus our attention on the first type. An elementary but paradigmatic example of such an analysis is provided by the algorithm that finds the largest element in an array of N numbers by sequential search. This appears as the first example of an algorithmic analysis in Knuth's *The Art of Computer Programming* [Section 1.2.10 of 18]; we shall call this program FindMax.

To set the stage for our discussion, it will be helpful to look at some of the main features of the analysis of FindMax, emphasizing how this analysis is typical of many others. Let there be N numbers stored in the array elements $X[1]$, $X[2]$, $\ldots$, $X[N]$. The program FindMax sets the variable M to the maximum of the $X[i]$'s by a left-to-right sequential scan:

```
$M \leftarrow X[1]$;
for $J$ from 2 to $N$ do
    if $X[J] > M$ then $M \leftarrow X[J]$ fi od.
```

We have now fixed the problem and algorithm to be studied; our next task is to choose a performance parameter of interest. The storage requirements of the program above are too simple to be interesting. Furthermore, the only subtle factor in its running time is the number of times the **if**-test comes out each way. Control will follow the TRUE branch of the **if**-test if and only if the element $X[J]$ for $2 \leq J \leq N$ breaks the record for the largest element seen so far. Such a record-breaking element is called a *left-to-right maximum*; for our purposes, it will be convenient to make the convention that the leftmost element is not a left-to-right maximum. We shall focus our analysis on counting the number A of left-to-right maxima.

Having chosen the performance parameter of interest, we have to decide what characteristic of that parameter's behavior to study. The usual choices are to analyze its value either in the worst case or in the average case. For the parameter A, the worst case analysis is no challenge. First, since the **if**-test is only performed once per execution of the **for**-loop body, the value of A can never exceed $N - 1$. Secondly, if the input array X is in increasing order, then every execution of the **if**-test will in fact come out TRUE; hence, the worst case value of A is exactly $N - 1$.

This worst case argument was too easy to reveal much structure, other than a tendency for exact worst case arguments to be composed of separate upper and lower bound proofs. But in general, analyses of worst case behavior often have a combinatorial flavor. The analyst's effort is directed at exploring a certain small and highly constrained collection of possible inputs, to discover which of them actually displays the worst performance.

For average case analyses, on the other hand, rather different kinds of arguments come up. Before we can talk about an average case, we must choose some probability distribution for the inputs to the algorithm; this defines what we mean by a random input. For sorting and searching problems, a convenient and not too unrealistic choice is the model in which the input is equally likely to be each of the $N!$ permutations of the set $\{1, 2, \ldots, N\}$; such an input is called a *random permutation*. Other input distributions are sometimes used in special situations, such as studying an algorithm's behavior on inputs with many repeated elements, but we shall be content with the random permutation model here.

The probabilistic distribution of the parameter A is given by the two-parameter family of numbers $\langle p_{a,n} \rangle$, where $p_{a,n}$ is the probability that A will take on the value a given that the size N of the input permutation takes on the value n. We can get some information about the numbers $p_{a,n}$ by the following argument. The final element $X[N]$ of the input array will be a left-to-right maximum if and only if it is the maximum of all the elements, that is, if and only if $X[N] = N$. This event happens with probability $1/n$. Whether or not this happens, the elements $X[1]$, $X[2]$, $\ldots$, $X[N-1]$ form a random permutation of the set $\{1, 2, \ldots, N\} - \{X[N]\}$. Furthermore, such a random permutation has just as many left-to-right maxima as a random permutation of the set $\{1, 2, \ldots, N-1\}$. This inductive insight determines the probabilities $p_{a,n}$ as the solutions of the recurrence relation

$$p_{a,n} = \frac{1}{n} p_{a-1,n-1} + \frac{n-1}{n} p_{a,n-1}$$

under appropriate initial conditions.

So far, we have expressed the probabilistic structure of A by means of a recurrence relation; we are left with the purely mathematical problem of studying the recurrence. This particular recurrence can be attacked with generating functions, which allow us to compute that the mean and variance of A are

$$\begin{aligned} \operatorname{mean}(A) &= H_n - 1 \\ \operatorname{var}(A) &= H_n - H_n^{(2)}. \end{aligned}$$

The details of this argument are given in Section 1.2.10 of Knuth [18].

The basic structure of the preceding argument is typical of many average case analyses of particular algorithms. Given an algorithm, a performance parameter of that algorithm, and a model of randomness for the input domain, we wanted to determine as precisely as possible the probabilistic structure of that performance parameter. By examining the program text and invoking our understanding of the underlying mathematical domain, we first interpreted the performance parameter A in terms of the underlying domain. This insight allowed us to find a recurrence relation that determined the distribution of A. Finally, we used standard mathematical methods to study the solution of that recurrence. In general, we shall call the first phase of such an analysis the *dynamic phase*; during the dynamic phase, we use our knowledge about such programming concepts as conditionals, loops, and recursion and our insight into the mathematical structure of the data domain to determine the distribution of the performance parameter under study in terms of a recurrence relation, integral equation, or other purely mathematical construct. In the second phase, which we shall call the *static phase*, we use whatever mathematical techniques are appropriate to investigate the solution of the recurrence that came out of the dynamic phase, either exactly or asymptotically.

Formalization.

What does it mean to formalize a mathematical proof? In one view, a mathematical proof is simply a convincing argument. Unfortunately, this simple viewpoint leads to various paradoxes. Partially in an attempt to eliminate these paradoxes, the field of mathematical logic has attempted to make the assumptions of arguments more explicit, and to restrict the reasoning steps permitted in arguments to a few elementary forms. These efforts culminated in the late nineteenth and early twentieth centuries with the development of the modern framework for mathematical logic [29]. In this framework, the statements that appear in mathematical proofs are encoded as strings of symbols over an alphabet, and the reasoning steps in proofs are modeled as transformations of these symbol strings. The small number of legitimate transformations can be studied very carefully, more carefully than could each of the many instances where those transformations are embedded in arguments. As a result, we can have greater confidence in those arguments that have been successfully encoded in terms of such a string manipulation. The rules that describe the legal strings of symbols and the rules for correctly manipulating those symbols together make up a *formal system.* The field of mathematical logic builds formal systems in which proofs can be encoded, and also studies these formal systems as mathematical objects in their own right.

Since set theory serves as the underlying basis for most of mathematics, formal systems for set theory have received particular attention. Currently, the von Neumann-Gödel-Bernays formalization of set theory is perhaps the most popular [29]. In principle, the proofs of classical mathematics could all be reduced to symbolic manipulations in this system, perhaps extended by suitable extra axioms, such as the Axiom of Choice. Furthermore, the Incompleteness Theorem of Gödel shows that it is impossible to do away with the occasional need for extra axioms. In particular, the Incompleteness Theorem shows that no formal system can be built whose theorems are precisely the true first-order statements about the integers under addition and multiplication.

Thus, the formalization of mathematical proofs is fairly well understood in principle. On the other hand, it is very rare in practice for anyone to actually attempt to carry out the formalization of non-trivial portions of classical mathematics. The details are complex and tedious enough to make such a formalization quite a formidable undertaking. With machine assistance now available, research is currently proceeding on this question. For example, a group at the Technological University of Eindhoven has constructed a formal system called AUTOMATH into which the classical book *Grundlagen der Analysis* by Landau has been completely translated [6, 32].

From one point of view, the analysis of algorithms is simply a part of classical mathematics. That is, it is straightforward in principle to construct a set theoretic model of an ALGOL machine, and hence to develop a rigorous mathematical definition of what it means to execute a program. Therefore, still in principle, the arguments that arise in the analysis of algorithms are really just disguised versions of complex symbol manipulations in a formal system for set theory. However, the translation of algorithmic analyses into set theory is sufficiently abstruse as to be of little practical value to the analyst. The sequence of actions dictated by a program often affects the state of the executing process in a rather subtle way. Although these effects can be encoded in set theory, there is something to be said for building a special purpose formal system instead, a system whose design incorporates the correspondence between program steps and process state. In fact, there is already in existence a large body of research concerning formal systems for reasoning about the properties of programs: the fertile area called *program verification.* It is high time that we give this area some consideration in our deliberations.

Program Verification.

The portion of program verification research that is most relevant to our current quest is the construction of formal systems that explicate the relationship between executable code and the static properties of process state. The word *process* here refers to an abstract entity that executes a program on particular input data. The *state* of a process is merely a vector containing the current values of the program variables; in particular, we will make the convention that the value of the program counter and the contents of the stack (if any) are not accessible parts of the process state. A typical program verification system is built on two formal languages. First, there is the executable code of the subject program, written in a programming language of some sort. Augmenting this, there is an assertion language, often closely resembling the first-order predicate calculus, in which certain properties of the state of a process can be described. The verification system then chooses some method for associating assertions about the process state with points in the program's flow of control. The most common choice is the *method of inductive assertions*, where the assertions are associated with textual locations in the program with the understanding that they are to hold whenever control passes that location. However programs and assertions are associated, we shall call the resulting structure of program and assertions together an *augmented program*; augmented programs are the well-formed formulas of a program verification system.

Besides these two formal languages, the other important component of a program verification system is a collection of syntactic mechanisms that allow one to derive certain well-formed formulas, that is, certain augmented programs, as theorems. If the formal system is sound, any augmented program that can be derived will in fact be correct. In systems based on the method of inductive assertions, correctness simply means that, for every possible path of control through the program from one assertion to another, the truth of the assertion at the beginning of the path implies the truth of the assertion at the end of the path. In systems that don't follow the inductive assertion paradigm, there is some other natural notion of correctness for an augmented program. When an augmented program has been shown to be a theorem of a sound system, this verifies a certain behavioral property of the program. Sometimes that property corresponds to a useful real-life characteristic: for example, the programs that solve certain simple tasks can be characterized by two assertions, the first of which may be assumed to hold upon input to the program, and the second of which is hoped to describe the corresponding output state. If a formal system can derive a theorem that contains a given program with these two assertions at its entry and exit, and possibly with other assertions in the middle, then that program has been shown to have the correct input-output behavior by formal manipulation.

The mechanisms of program verification systems vary over a fairly wide range, but many of them are based more or less closely on the ideas of Floyd and Hoare. Indeed, the Floyd-Hoare collection of techniques is so standard that they are normative for the field of program verification: each new idea is first compared with this standard, which we shall refer to as the Floyd-Hoare system for the verification of partial correctness. Floyd developed the ideas in the context of flowcharts [8], while Hoare concentrated instead on programs in a structured ALGOL-like language [13]. Since the latter techniques have proved more popular, we shall usually follow Hoare's lead rather than Floyd's.

We shall write the augmented programs of Hoare's system in the form $\{P\}S\{Q\}$. Here P, called the *precondition*, and Q, called the *postassertion*, are assertions about the process state, and S is an ALGOL-like program with a single entry and a single exit. The formula $\{P\}S\{Q\}$ is the formal system's encoding of the following concept: if a process begins to execute S in a state that satisfies P, and if this execution terminates normally, then the state of the process on exit from S will satisfy Q. The formula $\{P\}S\{Q\}$ does not imply that S will halt, and this is what is meant by the world "partial" in the phrase "partial correctness". A program S is called totally correct with respect to the assertions P and Q if it is partially correct with respect to them—that is, the formula $\{P\}S\{Q\}$ holds—and if it is also guaranteed to terminate normally whenever the input satisfies P. When a Floyd-Hoare system is supplemented by formal methods for verifying termination, it becomes a system for the verification of *total correctness.*

The symbol manipulations of a Floyd-Hoare system are designed to distinguish the correct augmented programs—formulas of the form $\{P\}S\{Q\}$—from the incorrect ones, or at least to allow most of the correct ones to be verified. Basically, these manipulations involve specifying how assertions move over program text. The legitimate manipulations in the system are described by *axiom schemata* and *rules*, and there will be at least one of these for each syntactic construct of the programming language. A formula in the system that has been justified from axioms by means of the rules is written $\vdash\{P\}S\{Q\}$ and called a *theorem*, in accordance with standard logical practice. The Rule of Composition is a simple example of a rule:

$$\frac{\vdash\{P\}S\{Q\},\quad \vdash\{Q\}T\{R\}}{\vdash\{P\}S;T\{R\}}.$$

The formulas above the horizontal line are called *premises*, and the formula below the line is the *conclusion.* If formulas matching the premises have been derived in the system, this rule allows the derivation of the conclusion. Such a rule provides one form of definition of the associated programming language construct; in fact, several programming languages have been formally specified in terms of the associated proof rules [14, 23].

The techniques of program verification have advanced to the point where many interesting properties of non-trivial programs can be demonstrated within such systems [27]. Indeed, a large part of the motivation for our effort to formalize algorithmic analysis comes from the success of program verification.

Scope of the Proposed System.

Program verification addresses the question of building formal systems in which the correctness properties of programs can be studied directly, without a clumsy translation back into set theory. The basic quest of this thesis is the construction of a formal system that addresses in a similar direct way some of the issues in the performance analysis of particular algorithms. Our immediate goal is to define more exactly what parts of algorithmic analysis we propose to formalize.

Recall that the analyses of particular algorithms basically fall into two classes: studies of the worst case, and studies of the average case. Considering the worst case analyses first, let us take the worst case analysis of FindMax as a motivating example. The upper bound portion of this analysis can be formalized in Floyd-Hoare correctness systems by the use of a *counter variable*; this approach was introduced by Knuth [exercise 1.2.1-13 in 18]. We can add to the program a new variable C, set initially to zero, and incremented exactly once each time that a new left-to-right maximum is found. The resulting program is

```
$C \leftarrow 0$; $M \leftarrow X[1]$;
for $J$ from 2 to $N$ do
    if $X[J] > M$ then $M \leftarrow X[J]$; $C \leftarrow C + 1$ fi od.
```

In a Floyd-Hoare system, we can then verify that the assertion $C \leq J - 2$ will hold at the beginning of the body of the **for**-loop. In particular, this demonstrates that the value of C will never exceed $N - 1$ on exit from the program, and hence gives us an upper bound on the worst case number of left-to-right maxima.

To show that $N - 1$ is actually a lower bound on the worst case as well, we must find an input that causes FindMax to run this slowly. As we mentioned earlier, an array in increasing order actually displays such worst case behavior. We could use Floyd-Hoare techniques to advantage for part of the lower bound argument as well. If we add to the input assumptions the assertion that the input array is in increasing order, then Floyd-Hoare techniques will allow us to show that the value of C upon exit from the program is exactly $N - 1$. In general, that is, Floyd-Hoare techniques address the question of tracing the program's behavior on the particular input that displays worst case performance. But that is not all of the lower bound proof; the other half is that we must demonstrate the existence of this input. Although it is clear that there exist arrays whose elements are in increasing order, we must check in general that the assertions that we are now assuming about the input are actually satisfiable. This part of the argument seems to belong to combinatorics more than anything else. At least, formal reasoning techniques that relate assertions and programs aren't really relevant, since the program no longer enters the picture. Summarizing, it seems that the portions of worst case arguments that deal with the program directly can be handled by standard program verification techniques. Therefore, we turn the attention of our quest to average case analyses.

Recall that a typical average case analysis can be divided into two phases, which we are calling *dynamic* and *static*. In the dynamic phase, the analyst uses information about what programs mean and how they behave to derive some form of recurrence that defines the probabilistic distribution of the performance parameter of interest. The static phase is then devoted to the solution of that recurrence. The static phase arguments are really independent of the algorithm; the recurrence is studied by purely mathematical means. Thus, formalizing the static phases of analyses will probably demand the same kinds of ideas and methods that are needed in formalizing the bulk of classical mathematics. Special purpose formal systems that know about programs would not be helpful.

But the dynamic phase of analyses is quite a different story, and it is here that our quest will be concentrated. The dynamic phase attempts to deduce a recurrence relation by applying knowledge about both mathematics and programming. For example, consider the dynamic phase of the average case analysis of FindMax. The final result of that effort is a recurrence that relates the number of left-to-right maxima in an n-element permutation to the number in an $(n - 1)$-element permutation. In some sense, this recurrence could be thought of as unwrapping the probabilistic effects of one execution of the body of the **for**-loop in the program, since that loop goes from a $(J-1)$-element permutations to an J-element one. There is thus some correspondence between the dynamic structure of the executing program and the static structure of the recurrence; this is one indication that an appropriate formal system might allow us to deduce that the program and recurrence really do correspond by means of relatively simple symbol manipulations. And the traditional derivation could use some formalization, since it throws around potent phrases such as "relative order of the remaining elements" fairly freely. These phrases are intuitively convincing, but certainly not completely formal.

This then we shall take as our quest: by suitably extending the concepts of Floyd-Hoare program verification, to build a formal system in which the dynamic phases of the average case analyses of at least some interesting algorithms can be encoded.

Value of the Proposed System.

It is worthwhile pausing for a moment to attempt to assess the benefits that such a formal system might have. One obvious candidate for such a benefit is the analog of the claimed benefits of program verification. The most frequently touted reward of a formal system for reasoning about the correctness of programs is the ability to produce software that is certifiably correct in some sense. That is, an argument has been presented in a certain very restricted way that justifies the correspondence between the executable code and certain assertion language specifications. Furthermore, if these specifications correspond to the real-life demands on the program, the program is then substantially more likely to perform its real-life job adequately. Since correct programs have tremendous real-life advantages over incorrect ones, any ability of the field of program verification to contribute to increased correctness is a powerful selling point.

Note, however, that the corresponding claim does not have quite the same appeal in the case of algorithmic analysis. Not that many algorithms have been analyzed even informally. And, although incorrect analyses have been published, the proliferation of incorrect analyses has not been a substantial problem to date. A program's efficiency is important in real-life situations, but, over a certain range, not nearly as critical as its correctness. Therefore, the fact that algorithmic analyses encoded in a formal system would be less likely to contain errors is only a weak inducement to build such a system.

Comparatively speaking, then, the fact that a formal system serves as an extra assurance of argument validity is a less compelling reason to formalize performance analyses than correctness arguments. The same factors that underlie this observation, however, also tend to insulate our current effort from some of the attacks that have been levelled at program verification [4]. People with the goal of large, verifiably correct software systems have been dismayed by the tendency of program verification efforts over the years to keep working on the same small and clean types of programs. Although the subtlety of the arguments that can be handled has increased, there is some question concerning the amount of progress in the direction of being able to handle the more elementary but very large, complex, and arbitrary programs that people are actually called upon to write. Some people view the progress of program verification in this direction of practicable applicability as disappointingly slow (Dijkstra is an exception [7]). Although certain restricted classes of arguments such as type checking are used throughout long programs, the formal systems seem to be better in general at short, subtle arguments than at long, straightforward, but complicated ones. The analysis of algorithms, on the other hand, restricted itself at the outset to dealing only with the cleaner problems and programs that arise in simple and well-behaved disciplines. Therefore, even if the critics of program verification research are right, building formal systems for algorithmic analysis might still make sense.

But if verifying the accuracy of analyses is not a compelling reason, why should we try to formalize the analysis of algorithms? One major benefit of such a formalization is a better understanding of the structure of the analyses as arguments. For example, consider the notion of random subfiles. Some sorting programs determine chunks of the input array on which to call themselves recursively; if these chunks of the input constitute random arrays at the time of the recursive call, then the sorting program is said to have *random subfiles.* Many sorting algorithms with random subfiles have been analyzed on the averge, but no substantial progress has yet been made on the average case analysis of any sorting program with non-random subfiles. Thus, the notion of random subfiles has important intuitive content to the algorithmic analyst. A formal system for algorithmic analysis could have impact on this notion in several ways. First, it might be possible to demonstrate by a metatheorem about the system that any sorting program with random subfiles will be analyzable by a certain technique, that is, to characterize what classes of sorts with random subfiles succumb to what analytic techniques. Secondly, the question of whether or not a sort has random subfiles essentially boils down to a question about the symbols of the program, the if-tests, the assignments, and the recursive calls. In the context of a formal system for algorithmic analysis, it might be possible to state interesting formal criteria that determine when a sorting program will or will not have random subfiles. A hope for this general kind of insight is perhaps the best motivating factor for our quest. Unfortunately, the system that we shall construct cannot handle recursive procedures, so formal insights into the notion of random subfiles must await a more powerful system.

A somewhat parallel situation currently pertains in mathematics itself. The field of mathematical logic contributed much to mathematics even before any efforts were begun to use formal systems to encode classical mathematics on a large scale. The primary contributions were clearer insights into the nature of mathematical proof, and a better sense for the basic assumptions upon which various fields depend. Gödel's Incompleteness Theorem and Cohen's proof of the independence of the Axiom of Choice are outstanding examples of logic's contributions. There is a good chance that similar insights into the structure of the arguments in the analysis of algorithms can also be achieved by studying appropriate formal systems. Recently, methods that grew out of mathematical logic and model theory, such as Non-Standard Analysis, are actually contributing to the development of classical mathematics. Perhaps it is too much to hope for, but there might be some computer science analog to Non-Standard Analysis somewhere out there waiting to be turned up.

Computers and Formal Systems.

Computer science has a rather special relationship to formal systems that we haven't yet given its due consideration: computer scientists work with and know about computers, which are symbol manipulators of historically unparalleled speed and accuracy. Any attempt to encode large or complex things in a formal system often leads to a situation where the details of performing the symbol manipulations, while elementary in principle, are too tedious to be carried out by people in practice. This is one situation where computers can be employed with good effect. Those subproblems in the formal system that lie in decidable subdomains can be programmed up on a computer, and the machine can spare us those tedious details. In particular, this is probably a major reason why efforts to formalize large bodies of classical mathematics awaited the advent of modern computers. We must keep in mind, of course, that computers cannot do all the work, since, for example, there is no algorithm that determines all and only the correct statements of elementary number theory.

This special relationship between computer science and formal systems is one of the factors contributing to the growth of program verification. Not only can we design formal systems for correctness verification, we can also put them onto a machine. In an unfortunate clash of nomenclature, the set of programs that implement the formal system on the machine are also called a *system*, specifically, a *program verification system.* We shall call these systems *programming systems* to avoid confusing them with formal systems. Current programming systems for program verification use symbol manipulation algorithms in combination with input from the user in an attempt to facilitate the details of program verification in the formal system.

This same development should be possible in the field of the analysis of algorithms. Once we have developed a formal system for the field, we could put that system onto a machine, and attempt to have the machine do as many of the straightforward details of the symbol manipulations as possible. This area of research might be called *automating the analysis of algorithms*, to distinguish it from our current quest, which is formalization rather than automation. One valuable result of the automation style of research might be a programming system with the mathematical knowledge of MACSYMA [28], augmented by some understanding of how the probability distributions of program variables are affected by executable code, such as comes out of the formal system that we are going to build. The resulting programming system might be just what the doctor ordered for the algorithmic analyst who is attempting to decide, for example, just how small the subfiles should be before a properly tuned implementation of Quicksort resorts to an insertion sort rather than a partitioning phase.

Prior Work on Formalizing Algorithmic Analysis.

Several programming systems directed at automating the analysis of algorithms have been built, and any such system inherently has some kind of a formal system for reasoning about the probabilistic behavior of programs buried within it. At one extreme are those systems that are willing to approximate the true behavior of the executing process by a Markov chain [2, 30]. With each conditional branch in the program, such a system associates a fixed probability, independent of the current state of the process, that the test will come out each way. This is often only a crude approximation to the truth, but the approximate results that come out of such an analysis might provide useful data for such clients as optimizing compilers. Since we are attempting to formalize exact theoretical analyses, we cannot afford to make the simplifying assumptions of constant and independent branching probabilities.

With a somewhat different approach, Jacques Cohen and Carl Zuckerman [3] built an interactive system for assisting an analyst in estimating the efficiency of a program. This system helped the analyst with the mechanical details of the algorithmic analysis, but left the difficult question of determining the branching probabilities to the user. In fact, Cohen and Zuckerman end their paper on this system by commenting that further research should be directed at relieving the user of this task, but that this would require the system to possess deductive capabilities similar to those required of programs that verify program correctness. Our quest can be viewed as a first step in response to this challenge, in that we are attempting to build the formal system in which this deductive reasoning could take place.

A third related effort in the direction of automating the analysis of algorithms was undertaken in the development of the PSI automatic programming system, by Cordell Green and his students [9]. This system is a knowledge-based approach to the automatic programming problem. One of the experts that contributes to the construction of the output program is an efficiency expert, written by Elaine Kant [17]; this algorithm uses task-specific knowledge about the problem domain to advise the PSI system about the relative efficiencies of various low-level data and control structures into which its very-high-level programs could be refined.

But none of the above three types of efforts addresss directly the formalization of algorithmic analysis as we currently understand it. The specific issue of the formal system seems to have been first considered by Ben Wegbreit, who worked from the same basic insight that we are using as our base. In fact, the abstract of his paper *Verifying Program Performance* [33], describes our quest in different words:

> It is shown that specifications of program performance can be formally verified. Formal verification techniques, in particular, the method of inductive assertions, can be adapted to show that a program's maximum or mean execution time is correctly described by specifications supplied with the program. To formally establish the mean execution time, branching probabilities are expressed using inductive assertions which involve probability distributions. Verification conditions are formed and proved which establish that if the input distribution is correctly described by the input specification, then the inductive assertions correctly describe the probability distributions of the data during execution. Once the inductive assertions are shown to be correct, branching probabilities are obtained and mean computation time is computed.

In that paper, Wegbreit gives an analysis of the sorting program InsertionSort in his formal system. His sytem will serve as the starting point for our own formal system construction effort, which begins in the next chapter.

Recently, Dexter Kozen published a paper called *Semantics of Probabilistic Programs* [22] that addresses some of the same issues that we shall be considering, but from a somewhat different angle. Kozen's paper addresses the question of providing a formal denotational semantics for a class of probabilistic programs, that is, programs that are allowed to make random choices during the course of their execution. As it turns out, the framework that we shall develop for formalizing the probabilistic analyses of deterministic algorithms is also a natural framework in which to consider probabilistic algorithms. Kozen independently arrived at essentially the same framework that we shall propose. He then demonstrated a formal semantics for probabilistic **while**-programs in this framework, and related this semantics to an established field of mathematics, linear operators on Banach spaces, that has been explored for its own sake. We shall draw on Kozen's work as on Wegbreit's in what follows.

Before our literature survey can be called complete, we should also give credit to Arne T. Jonassen and Donald E. Knuth for their work in the paper *A Trivial Algorithm whose Analysis Isn't* [16]. They considered the folowing problem: Take a random binary search tree containing two keys; choose a new key at random, and insert it into the tree; then, choose one of the three keys currently in the tree at random, and delete it; finally, repeat this insertion-deletion process indefinitely. Inserting into a binary search tree is a straightforward process, but deleting from one is trickier. In particular, deleting a non-leaf from a tree demands some reshuffling of the nodes. The probabilistic structure of the search trees resulting from this insertion-deletion regimen turns out to be quite subtle. In the dynamic phase of their analysis, the first seven pages of the paper, Jonassen and Knuth reduce the problem to a set of integral equations, the continuous analog of recurrence relations. The solution of these integral equations in the following static phase fills the remaining fifteen pages with fairly hairy mathematics: the solution involves Bessel functions.

The interesting thing about this paper from our current perspective is the level of reasoning used in the dynamic phase. The problem was sufficiently subtle that Jonassen and Knuth were forced to derive the recurrences in an almost mechanical fashion, working line by line from the associated program. Because this dynamic phase is carried out at such a low level, it serves as an excellent motivating example for those, like ourselves, who are attempting to formalize the dynamic phase of algorithmic analysis.

Looking Ahead.

In Chapter 2, we shall begin constructing a formal system by considering several alternative ways in which to phrase assertions that describe the probabilistic state of an executing process. To keep things simple, we restrict ourselves in Chapter 2 to loop-free programs. The complications introduced by allowing loops back into our programs are the subject of Chapter 3.

In Chapter 4, we shall cover in some detail the structure of one formal system, called the *frequency system*, for the dynamic phase of average case algorithmic analysis. After describing the assertion language of the frequency system, we shall discuss its axiom schemata and rules, and demonstrate their soundness in terms of Kozen's semantics. Chapter 5 then presents several examples of the use of the frequency system. The two major examples are the dynamic phases of the analyses of FindMax, which we discussed above, and of InsertDelete, which implements the repeated insertions and deletions in small binary search trees studied by Jonassen and Knuth.

Finally, in Chapter 6, we shall discuss ways in which the frequency system could be extended to handle other control structures. After extending the frequency system to handle the **goto**-statement, we shall pause to consider the dynamic phase of the analysis of InsertionSort, which allows us to compare Wegbreit's system and the frequency system in action. Chapter 6 closes by outlining several directions in which future research on formalizing the analysis of algorithms should proceed.

Chapter 2. Probabilistic Assertions for Loop-Free Programs

The Search for an Assertion Language.

We shall begin the construction of our formal system by thinking about what an assertion language that can describe the probabilistic structure of a process state might be like. We shall be forced to choose our assertion language from among several possibilitites, and, in the process of exploring these choices, we shall develop a model of programs and their executions called the *chromatic plumbing metaphor* that will serve as a helpful framework for our assertion language design. The chromatic plumbing metaphor also suggests a novel view of program semantics, which is exactly the view suggested by Kozen.

As our starting position, we begin with the insight that program variables should be considered to have distributions, just like the random variables of probability theory; they inherit these distributions from our chosen probability distribution on the input. Our job is to trace how these distributions are affected by the execution of code. We also start off with the sense that a probabilisitic assertion should give us some kind of information about these distributions. The current distributions of the program variables and their interrelationships can be thought of as forming the *probabilistic state* of the process; this notion makes sense only when the executions of the program on all possible inputs are considered simultaneously, with their associated probabilities. When the program is considered as acting on a single input, the program variables will have a unique value at each moment. These values make up what we used to call simply the *state* of the process, and shall now call its *deterministic state.*

It is helpful to keep in mind the analogy with the Floyd-Hoare situation. A Floyd-Hoare system deals with only one execution of the program at a time, and hence deals only with questions about the deterministic state. An assertion in such a system gives some information about the deterministic state of the executing process, although it need not characterize that state completely. In fact, with each assertion in a Floyd-Hoare system we can associate a certain set of deterministic states called the *characteristic set* of the assertion. The characteristic set of the assertion P is simply the set of all deterministic process states for which that assertion holds.

Of course, a Floyd-Hoare assertion is also a string of symbols, and it has a rather different structure when viewed from this perspective. First, there is a class of *atomic assertions* that describe an elementary characteristic of the process state, such as the formula $K = 1$. These atomic assertions are then combined with the *logical connectives* "and", "or", and "not", written $\wedge$, $\vee$, and $\neg$ respectively, and the *quantifiers* "for all" and "there exists", written $\forall$ and $\exists$. Note that each connective corresponds to an operation of elementary set theory on the associated characteristic sets. Taking the "and" of two assertions corresponds to taking the intersection of their characteristic sets; the "or" corresponds to the union; and "not" corresponds to set complement. Furthermore, the quantifier "for all" is a generalized, indexed version of "and" from a set theoretic point of view, allowing the specification of a more general form of intersection; similarly, "there exists" is a generalized, indexed form of "or". Thus, the connectives and quantifiers of a Floyd-Hoare system correspond to the elementary operations of set theory on the characteristic sets.

By analogy with the Floyd-Hoare situation, we shall attempt to construct our assertions about probabilistic state by combining certain atomic probabilistic assertions with certain connectives. Before we tie these ideas down any further, it will be helpful to introduce the chromatic plumbing metaphor for programs and their executions.

Chromatic Plumbing.

For now, we shall take as our programming language an ALGOL-like language without procedures, but we shall think of these programs in terms of their flowcharts. Take the flowchart of the subject program, and turn it into a plumbing network. The lines of the flowchart are the pipes of the network, which serve to guide the executing process from instruction to instruction. The process itself is modeled as a pellet that travels through the pipes; thus, the current position of the pellet in the plumbing network corresponds to the current location of control in the program, the current value of the program counter if you will. At **if**-tests in the program, the piping forks. A pellet coming down the in-branch of a fork takes either the left or the right out-branch of the fork depending upon whether the state of that process does or does not satisfy the **if**-test. The state of a process is simply the vector of values of the program variables. We shall model such a state by considering the control pellet to be colored, one color corresponding to each possible state that the process might be in.

Besides forks that come from **if**-tests, the plumbing network will contain three other types of features. Where different paths of control come together in the program, as at the end of an **if**-statement, there will be a join in the network. If the control pellet comes down either of the two in-branches of the join, it will continue down the single out-branch. The program also contains assignment statements. Since an assignment statement is the mechanism that changes the process state, we shall model assignment statements in the plumbing network by structures that might be called *repainting boxes.* When the control pellet reaches such a box, it is given a new coat of paint, whose color reflects the new state of the process. The new color is chosen as some function of the old color just as the new state after an assignment statement is some function of the old state of the process. Finally, the **start** and **halt** instructions in the flowchart turn into an input funnel and one or more output chutes in the plumbing network. We shall assume that one of the output chutes is distinguished as the *normal output chute,* since our flowcharts are modeling programs with a single entry and single exit.

Suppose that we have built the plumbing network corresponding to our program of interest. Executing the program on any particular input can then be modeled as follows: take a pellet and color it whatever color corresponds to the chosen input state. Then, drop this pellet into the input funnel of the plumbing network, and watch what happens. It will travel around the pipes, getting repainted by assignment boxes, and getting sent the appropriate direction by **if**-tests. If the program has loops, the pellet might very well pass one point in the plumbing network more than once. Eventually, the pellet may come dropping out of some output chute, corresponding to the termination of the program; the color of the pellet when it emerges will correspond to the state of the process upon termination. Or, if the program does not halt for this particular input, the pellet will continue to meander through the plumbing network forever.

We can use the chromatic plumbing metaphor to describe the kinds of results that the standard formal systems for program verification can handle. Floyd-Hoare logics provide results about a program's partial correctness. These are theorems of the form, "If a green pellet is dropped into the input funnel, and if that pellet ever comes out the normal output chute, then it will be blue when it does come out," or, "A pellet of a primary color dropped into the input funnel will be black if and when it ever comes out of the normal output chute." In particular, a partial correctness result does not guarantee the termination of the program; verification systems that deal with partial correctness are said to be using a *weak logic*. By contrast, total correctness theorems do make guarantees about termination; such systems are said to be using a *strong logic*. In our chromatic plumbing metaphor, a typical total correctness result would be, "If an orange pellet is dropped into the input funnel, it will eventually emerge from the normal output chute colored pink."

Probabilistic Chromatic Plumbing.

In the average case analysis of algorithms, we have to study more than just one execution of the program. In fact, we have to worry about the behavior of the program on all possible inputs, and about the probabilities of the various behaviors. This demands some modification of the chromatic plumbing metaphor. To keep things simple at the outset, let us first restrict ourselves to considering only loop-free programs, that is, programs whose flowcharts are directed acyclic graphs. This assumption simplifies things considerably, because each control pellet in loop-free programs can pass each point in the plumbing network at most once.

Now, in the domain of loop-free programs, our task is to modify the chromatic plumbing metaphor so that it can describe all possible executions of the program at once, with their associated probabilities. One natural way to achieve this modification is to allow more than one pellet to exist, where each pellet will model one possible execution of the program. In addition to its color, which corresponds to the state of the process as before, each pellet will also have an associated weight, which will be proportional to how likely this particular execution is, in comparison with other executions. That is, the events described by heavy pellets are more likely, or happen more frequently, than those described by light pellets.

Given this idea of weighted pellets, we can imagine constructing a bag of pellets that will model any chosen input distribution to an algorithm. An input distribution is really a collection of possible process states with their associated probabilities. Such a structure incorporates distributions for all of the program variables, and shows how those distributions are interrelated; thus, we hereby clarify the term *probabilistic state* by redefining it to mean a set of possible deterministic process states with associated probabilities. To construct a bag of pellets that models a chosen probabilistic state, we simply take one pellet for each possible deterministic state, and color it whatever color corresponds to that state; then we also adjust its weight to be proportional to the probability of that deterministic state. On the other hand, any bag of weighted pellets corresponds to a certain probabilistic state in a natural way, as well.

The chromatic plumbing metaphor with multiple weighted pellets can model the execution of an algorithm on all possible inputs at once. We begin by constructing a bag of pellets to model the chosen probabilistic input state. Then, we empty this bag of pellets into the input funnel of the plumbing network, and let each pellet independently travel through the network according to the old rules. Each pellet gets steered by **if**-tests, repainted by assignments, and (since our programs are currently assumed to be loop-free) eventually emerges from an output chute. But the actions of the various pellets are completely independent, and the weights of the pellets don't change during their trip through the network.

If we concentrate at any particular point in the plumbing network, and consider all of the pellets that ever pass that point, the colors and weights of those pellets describe the probabilistic structure of what happens at that point in the program. For example, if we normalize the weights of the pellets so that the total weight of the input bag is 1, then the total weight of all the pellets that pass any point in the network is just the probability that control will pass that point during a random execution of the program. Instead of focusing at that point in the network ourselves, we might as well postulate a little *demon* who sits on the pipe at the chosen point, and keeps track of the weights and colors of the pellets that go by. Since we are only interested in the weight totals rather than in how these totals are made up, we shall specify that the demon in fact just reports to us, for each possible color, the total weight of all of the pellets of that color that pass by. A complete set of reports from demons located at every possible spot on the plumbing network gives one kind of probabilistic description of the behavior of the algorithm executing on a random input. Therefore, thinking about these demon reports gives us one possible concrete sense for what an assertion about probabilistic state should do: it should partially specify a demon's report.

The report of a demon located somewhere in a program can be useful from a practical as well as a theoretical point of view. Monitoring devices conceptually similar to demons are provided by some programming language systems [15, 31].

Probabilistic Assertions.

An assertion in a Floyd-Hoare system is a partial description of deterministic process state. To prevent nomenclatural confusion, we shall reserve the word *assertion* in the future to refer to the probabilistic version; the term *predicate* will be used to denote an assertion of a Floyd-Hoare system. Our next task is to decide what an assertion should be, in our brave new probabilistic world. We shall continue to restrict ourselves to loop-free programs for a while, and to employ the chromatic plumbing metaphor with multiple weighted pellets.

Predicates in a Floyd-Hoare system are built up by combining certain atomic predicates with logical connectives. Guided by this analogy, we shall decide on certain elementary *atomic assertions*, and then allow ourselves to glue atomic assertions together with connectives. The obvious candidates for connectives in the probabilistic world are the same connectives that show up in Floyd-Hoare predicates: the connectives "and" and "or", along with their indexed generalizations "for all" and "there exists", and the connective "not". Those connectives will satisfy us for a while; but what is an atomic assertion to be?

For this question, the analogy with Floyd-Hoare systems isn't much help. But our general intuition is that an assertion is supposed to give us some information about how the program variables are behaving as if they were really random variables in the sense of probability theory. Therefore, one obvious candidate for an atomic assertion is a formula for the probability of an event, such as $\Pr(K = 1) = \frac{1}{2}$. This assertion describes the set of all probabilistic states in which the program variable K takes on the value 1 with probability precisely $\frac{1}{2}$. To be more precise, remember that our assertions are supposed to be partial desciptions of demon reports. Thus, this formula really asserts that half of all of the pellets by weight should correspond to deterministic states in which the predicate $K = 1$ holds.

We shall use this example as a guide for our first cut at what an atomic assertion should be: An atomic assertion is a formula of the form $\Pr(P) = e$, where P denotes an arbitrary predicate, and e denotes some kind of recipe for a real number, a real-valued expression in some language. The predicate P describes a particular set of deterministic states, and thus, it also determines the corresponding set of colors. The formula $\Pr(P) = e$ will be deemed to hold of a demon's report if and only if the total weight of all of the pellets of colors satisfying P, when divided by the total weight of all of the pellets of all colors, gives the quotient e. Let $\mathrm{Mass}(Q)$ for an arbitrary predicate Q denote the total weight of all the pellets of colors satisfying Q in the demon's report; we then have the relation

$$\Pr(P) = \frac{\mathrm{Mass}(P)}{\mathrm{Mass}(\textsc{true})},$$

since $\mathrm{Mass}(\textsc{true})$ will give the total reported weight of all of the pellets of all colors.

The Leapfrog Program.

To decide whether or not our current first cut at a definition for the concept of atomic assertion is any good, we shall now consider an example, the Leapfrog program:

$$\text{Leapfrog:}\quad \textbf{if } K = 0 \textbf{ then } K \leftarrow K + 2 \textbf{ fi}.$$

Suppose that a process begins to execute the Leapfrog program in a probabilistic state in which the variable K takes on the values 0 and 1 with equal probability. Note first that we can describe this initial state by an assertion that is a conjunction of two atomic assertions:

$$\left[\Pr(K = 0) = \tfrac{1}{2}\right] \land \left[\Pr(K = 1) = \tfrac{1}{2}\right].$$

If K is the only component of the process state, then this assertion completely defines the probabilistic state, since the two events it describes are mutually exclusive, and their associated probabilities sum to one. On the other hand, if there are other program variables floating around as well, then this assertion gives only partial information about the probabilisitic process state. Assuming that K is the only program variable for simplicity, we could make up this input state as a bag of pellets by taking two pellets of equal mass, and coloring one of them the $K = 0$ color, and the other the $K = 1$ color.

Intuitively, the net result of the execution of the Leapfrog program will be to take the probabilistic mass associated with the condition $K = 0$, and move it over to the condition

$K = 2$; this explains our choice of the name "Leapfrog". We can trace Leapfrog's execution on our chosen probabilistic input by thinking about the associated plumbing network. The two pellets of equal mass will arrive first at the **if**-test. The $K = 0$ pellet will continue down the TRUE out-branch of this test, while the $K = 1$ pellet will continue down the FALSE out-branch. The $K = 1$ pellet emerges unchanged from the output chute of the whole program; the $K = 0$ pellet passes through the assignment box corresponding to $K \leftarrow K + 2$, which repaints it the $K = 2$ color, and then emerges from the output chute.

With our current definition of "assertion", what can we assert about this program's execution? We already characterized the input state by an assertion above. Consider next the two out-branches of the **if**-test. On the TRUE out-branch, the predicate $K = 0$ will hold with certainty, so the appropriate assertion is $\Pr(K = 0) = 1$; similarly, on the FALSE out-branch, the natural assertion is $\Pr(K = 1) = 1$. After the assignment box on the TRUE branch, we have the condition $\Pr(K = 2) = 1$. And finally, the output is described by an assertion very similar to the input assertion:

$$\left[\Pr(K = 1) = \tfrac{1}{2}\right] \wedge \left[\Pr(K = 2) = \tfrac{1}{2}\right].$$

All of these assertions are valid statements about the corresponding demon reports; each right-hand side gives the percentage of pellet mass that satisfies the left-hand side predicate.

Next, let us consider each feature in the flowchart locally, and think about the assertions in the neighborhood of that flowchart feature. Consider first the **if**-test; coming into a test on the predicate $K = 0$, we have some probabilistic state that satisfies the assertion

$$\left[\Pr(K = 0) = \tfrac{1}{2}\right] \wedge \left[\Pr(K = 1) = \tfrac{1}{2}\right].$$

Note that we can deduce the appropriate assertions for the out-branches of the fork purely by local formal reasoning; they must be $\Pr(K = 0) = 1$ and $\Pr(K = 1) = 1$ on the TRUE and FALSE out-branches respectively. In fact, the job of going from the probabilities before the fork to the probabilities after the fork simply corresponds to taking conditional probabilities. Similarly, we could imagine adjusting the TRUE branch assertion to be $\Pr(K = 2) = 1$ after the assignment by purely local reasoning.

But the final join is quite a different story. Coming into the join from the TRUE branch, we have some probabilistic mass in which the program variable K has the value 2 with certainty; the FALSE branch contributes some mass in which K has the value 1 with certainty. That is all that we can deduce from the local assertions, and it is not enough to determine the probabilistic structure of K after the join. Indeed, the local information about the input to the join only allows us to conclude that the assertion

$$\Pr\bigl([K = 1] \vee [K = 2]\bigr) = 1$$

holds, and this is a substantially less informative statement than the output assertion given above. In fact, this output assertion corresponds to drawing the Floyd-Hoare style conclusion that, if

K is 1 on one in-branch of a join and K is 2 on the other, then K is either 1 or 2 on the out-branch; there is nothing probabilistic about this reasoning at all.

The problem is that the assertions on the in-branches of the final join don't describe the relative probabilities of entering the join from each of the two in-branches. In our example, control is equally likely to enter the final join from either of the two in-branches, since the two pellets of the original input were equally heavy. But this fact is based on information that is not reflected in the local assertions; in fact, we explicitly threw this information away when we computed the conditional probabilities at the **if**-test. We shall immortalize this difficulty by naming it the *Leapfrog problem.*

People who are used to Floyd-Hoare program verification might be tempted to claim that, since the assertions $\Pr(K = 2) = 1$ and $\Pr(K = 1) = 1$ hold respectively on the TRUE and FALSE in-branches of the final join, the assertion

$$[\Pr(K = 2) = 1] \vee [\Pr(K = 1) = 1]$$

should hold on the out-branch of the final join. Wrong! A probabilistic assertion, remember, is a partial description of a demon's report. This assertion specifies that the demon either reports that "All the mass that went by was colored the $K = 2$ color," or that "All the mass that went by was colored the $K = 1$ color." In fact, the output demon of the Leapfrog program does not give either of these reports, but rather gives a report that is halfway between the two. In the Floyd-Hoare world, it is legitimate to describe the out-branch of a join by the "or" of the predicates that describe the in-branches; but this technique doesn't work in the probabilistic world.

The assertions in Ben Wegbreit's formal system [33] are a combination of Floyd-Hoare predicates with the $\Pr(P) = e$ style of atomic probabilistic assertions that we are currently considering. Therefore, Wegbreit's system would have difficulty with the Leapfrog problem. The fact that Wegbreit's system is powerful enough to handle InsertionSort is some manner shows that even systems with no cure for the Leapfrog problem can be quite powerful. We shall strive for a solution of the Leapfrog problem, however.

Frequencies instead of Probabilities.

One way to avoid the Leapfrog problem is to avoid the rescalings that are associated with taking conditional probabilities. In this scheme, assertions measure a quantity that is like probability in every way except that it does not always have to add up to 1. We shall call this quantity *frequency*, and our next job is to adjust the chromatic plumbing metaphor to deal with frequencies.

Suppose that the pellets of the chromatic plumbing metaphor have weights that are expressed in an explicit unit of measure, say *grams.* We shall usually normalize this unit so that the total weight of all of the pellets in the input bag is exactly 1 gram. With this normalization, the weight of any pellet is a measure of the frequency of the event that the pellet describes, in the following sense. Suppose that we repeatedly perform random and independent executions of the program. And suppose, for example, that our chosen probabilistic input state contains a yellow pellet of mass $1/k$ grams, for some integer $k \geq 2$. On the average, once out of every k times that we execute the program, the actual deterministic input that occurs will be the input state that corresponds to the color yellow. All along the path that the originally yellow pellet travels in the network, the demons will report a mass of $1/k$ grams for whatever color the pellet has been repainted. The fact that the pellet weighs $1/k$ grams means that it represents, on the average, one out of every k executions of the program. In some sense, the weight of any pellet measured in grams is equal to the execution frequency of the corresponding event measured in "expected times per execution of the whole program". The weights that the demons report back will now be expressed in grams, and we hereby rename the Mass function defined earlier to be the "Fr" function, where "Fr" stands for "frequency" just as "Pr" stands for "probability".

We can turn this idea of measuring weights in grams into a solution of the Leapfrog problem by changing our definition of atomic assertions so that they also measure grams. Define an atomic assertion to be a statement of the form $\mathrm{Fr}(P) = e$, where P is a predicate and e is a real-valued expression. This type of atomic assertion correctly describes a demon's report if and only if the total weight of all of the pellets reported, of colors that satisfy P, is precisely e; there is no rescaling, no dividing by the total weight of all of the pellets. We shall call these formulas *frequentistic atomic assertions*, to contrast them with our earlier *probabilistic atomic assertions.*

We should also clarify our terminology for states. Recall that a *probabilistic state* is a collection of deterministic states with associated probabilitites. We can model a probabilistic state by a bag of pellets of arbitrary total weight, since it is only the ratios of the weights of the various pellets that are critical. Thus, a probabilistic state really corresponds to an equivalence class of bags of pellets, where two bags are equivalent if they are rescalings of each other. By contrast, we now define a *frequentistic state* to be a collection of deterministic states with their associated frequencies. Note that a demon's report is simply a complete description of a frequentistic state: it associates a definite weight in grams with every possible deterministic state. We can put our frequentistic atomic assertions together with connectives to form *frequentistic assertions*, and each of these will have an associated *characteristic set* that is precisely the set of all frequentistic states in which it holds.

We can now go through and see how the Leapfrog program looks in this new format. The input assertion is the same as before,

$$\left[\mathrm{Fr}(K=0)=\tfrac{1}{2}\right] \wedge \left[\mathrm{Fr}(K=1)=\tfrac{1}{2}\right].$$

At the **if**-test on the predicate $K = 0$, this total of 1 gram of execution mass splits, with half of it continuing down the TRUE out-branch, and the other half continuing down the FALSE out-branch. On the TRUE out-branch, we have the assertion

$$\left[\mathrm{Fr}(K=0)=\tfrac{1}{2}\right] \wedge \left[\mathrm{Fr}(K\neq 0)=0\right].$$

The second atomic assertion here is rather a surprise, but note that we can't get by without it. The assertion $\mathrm{Fr}(K = 0) = \frac{1}{2}$ tells us that one half of a gram of execution mass goes by in which K has the value $\mathbf{0}$, but it doesn't eliminate the possibility that other execution mass goes by in which K is not $\mathbf{0}$. We can eliminate that possibility either by, as above, adding the condition $\mathrm{Fr}(K \neq 0) = \mathbf{0}$, or by adding the condition $\mathrm{Fr}(\text{TRUE}) = \frac{1}{2}$; the former course seems the more natural. In fact, to be cautious, it would probably be a good idea to add a similar extra condition to the input assertion, and replace our earlier version by

$$\left[\mathrm{Fr}(K = 0) = \tfrac{1}{2}\right] \wedge \left[\mathrm{Fr}(K = 1) = \tfrac{1}{2}\right] \wedge \left[\mathrm{Fr}([K \neq 0] \wedge [K \neq 1]) = \mathbf{0}\right].$$

We stated that it would be our convention to normalize the gram so that the total mass of the original input bag of pellets was exactly one gram, but it is safer not to build that convention too deeply into our reasoning.

The FALSE out-branch of the **if**-test gets the symmetric assertion

$$\left[\mathrm{Fr}(K = 1) = \tfrac{1}{2}\right] \wedge \left[\mathrm{Fr}(K \neq 1) = \mathbf{0}\right],$$

and the TRUE out-branch after the assignment is described by

$$\left[\mathrm{Fr}(K = 2) = \tfrac{1}{2}\right] \wedge \left[\mathrm{Fr}(K \neq 2) = \mathbf{0}\right].$$

Finally, reasoning from these two assertions alone, we can deduce the appropriate assertion to put on the out-branch of the final join. In particular, we add together the frequencies of events contributed by each of the two in-branches, and get

$$\left[\mathrm{Fr}(K = 1) = \tfrac{1}{2}\right] \wedge \left[\mathrm{Fr}(K = 2) = \tfrac{1}{2}\right] \wedge \left[\mathrm{Fr}([K \neq 1] \wedge [K \neq 2]) = \mathbf{0}\right].$$

The success of this last step shows that measuring frequencies rather than probabilities is indeed one way of avoiding the Leapfrog problem.

For a while, we won't take a definitive position on the question of frequencies versus probabilities. Instead, we shall change our focus of concentration somewhat, and worry about the connectives with which atomic assertions are put together.

The Arithmetic of Frequentistic States.

So far in our development of a probabilistic assertion language, we have been satisfied with borrowing the logical connectives that are used in Floyd-Hoare systems, which are really the connectives of the predicate calculus. But consider the final join of the Leapfrog program once again. When we combined the two frequentistic assertions on the in-branches into a single frequentistic assertion for the out-branch, we were doing something that was much more like addition than like any of the logical connectives. This suggests that there is some arithmetic structure to the set of all frequentistic states that it would be profitable to explore.

To get a handle on this structure, it is helpful to think in terms of the chromatic plumbing metaphor: recall that a demon's report is simply a description of a frequentistic state. Suppose that c and d represent the reports of demons on the two in-branches of a join in the network.

From c and d, we can compute what a demon located on the out-branch of that join must report. In particular, the out-branch demon should report for each possible deterministic state a weight that is the sum of the weights ascribed to that state by c and d. It is natural to call this frequentistic state $c+d$. The set of all possible frequentistic states is closed under this addition operation; it is also closed under multiplication by nonnegative scalars. Thus, it has a lot of the structure of a vector space. But it has other structure as well; note that there is also a natural partial order. The relation $c \leq d$ holds between two frequentistic states if and only if the frequency that c assigns to each event does not exceed the frequency assigned by d.

If joins in the plumbing network correspond to a natural operation on the frequentistic states, we should next think about forks. Suppose that a frequentistic state c is reported by the demon on the in-branch of a fork. Let P denote the test associated with this fork. In programming terms, P is a side-effect free boolean expression; but for our current purposes, we shall consider P to be merely some predicate. Thus, P corresponds to some division of the set of all deterministic states into two classes, those where P does and does not hold. Now, what will a demon on the TRUE out-branch of this fork report? We shall denote this report by $c \mid P$, which might be read "c restricted to the truth of P". If a deterministic state statisfies the predicate P, then $c \mid P$ ascribes the same mass to that state that c ascribed; if a deterministic state does not satisfy P, then $c \mid P$ ascribes it a mass of zero. Symmetrically, a demon on the FALSE out-branch of the fork will report the frequentistic state $c \mid \neg P$, which is c restricted to the falsity of P. The restriction and addition operations are related by the identity $c=(c \mid P)+(c \mid \neg P)$, which essentially states that the program

if P **then nothing else nothing fi**

really is a no-op.

The restriction and addition operations allow us to record the effects upon frequentistic states of forks and joins in the plumbing network. In our current domain of loop-free programs, we will have handled all of the constructs of a program if we can determine the effect that an assignment has upon frequentistic state. An assignment is simply a function from the deterministic input state to a deterministic output state. Given a frequentistic input state, we can find the resulting frequentistic output state by applying this function to each possible deterministic input state, multiplying by the corresponding frequency, and summing. To put this another way, an assignment can be thought of as a linear function from the set of all frequentistic states to itself.

In fact, there are more linear functions floating around as well. Consider what a program really is from our current point of view. Through the chromatic plumbing metaphor, a program corresponds to a plumbing network that maps input bags of pellets into output bags of pellets, or, to put it another way, input frequentistic states into output ones. And this mapping will have to be linear. Standing back a little, we get the somewhat surprising sense that there just might be some real linear algebra going on. In particular, it might be possible to adjust our definitions so that the set of all frequentistic states really would be a vector space, and so that the meaning of a program could be defined to be some (continuous?) linear transformation of this space. This vague idea is given solid substance in Dexter Kozen's paper.

Kozen's Semantics for Probabilistic Programs.

Dexter Kozen has attacked the problem of providing a semantics for probabilistic programs, that is, for programs that are allowed to make random choices. The semantics that he developed turns out to be based precisely on the concepts that we arrived at in the last section, in our consideration of the probabilistic analyses of deterministic programs. This coincidence is not as surprising as it might seem at first glance. Suppose that a program makes random choices. We can modify the program slightly to eliminate the random choices by extending the program's input to include a file of random variables. Whenever the original program would have made a random choice, the modified program can merely examine the next random input variable, and act accordingly. This transformation shows that there is only a fine dividing line between those programs that make random choices and those that take random inputs.

We are going to adopt Kozen's semantics as the basis for our further efforts at formal system construction, so we shall proceed to sketch the main results here; the construction is given in more detail in Kozen's paper [22]. There is a real need for a more precise development of a probabilistic semantics. So far, our arguments have been based on our intuitions about program behavior, and that is a fine start. It is roughly true that a frequentistic state is a collection of deterministic states, together with associated frequencies. This level of definition would suffice if, for example, there were only a finite set of possible deterministic states. For most programs, however, there are at least an infinite and often an uncountable number of possible deterministic states. And this demands a more careful definition. We now summarize Kozen's definitions and results.

Each program variable will have an associated data type D; we will use the same symbol D to denote the set of all values of that type. Kozen begins by assuming that with each basic data type D, there is an associated σ-algebra [10]; for the integers, the natural σ-algebra is the power set of the integers, while, for the real numbers, it is the σ-algebra of all Lebesgue measurable sets. The choice of a σ-algebra makes each data type D into a measurable space, which we shall also denote D.

The deterministic state of a process is the vector of values of the program variables. If there are n program variables $X_1, X_2, \ldots, X_n$ of types D_1, D_2, $\ldots$, D_n respectively, then the set of all ~~possible~~ deterministic states $\mathfrak{S}$ is the Cartesian product of the basic types

$$\mathfrak{S} = D_1 \times D_2 \times \cdots \times D_n.$$

We shall associate with $\mathfrak{S}$ the smallest σ-algebra that contains all rectangles. This makes $\mathfrak{S}$ into a measurable space.

A *measure* is a countably additive, real-valued set function on a measurable space; in particular, we assume at this point that all of the values of a measure are finite. A measure is called *positive* if its values are all nonnegative. We can define a *frequentistic state* more precisely as a positive measure on $\mathfrak{S}$. The set $\mathfrak{F}$ of all measures on $\mathfrak{S}$ forms a real vector space.

Furthermore, there is a natural norm for this vector space; we define the norm $\|c\|$ of a measure c to be the mass that the absolute value of c ascribes to the entire space $\mathfrak{D}$. For positive measures, this norm is simply given by the formula $\|c\| = c(\mathfrak{D})$. The vector space $\mathfrak{F}$ forms a Banach space under this norm. The set of all positive measures, which we shall write $\mathfrak{F}^+$, is the *positive cone* of $\mathfrak{F}$. Furthermore, this positive cone defines a partial order on $\mathfrak{F}$, under which it becomes a conditionally complete vector lattice, and even an (L)-space.

Note that a frequentistic state is merely a point in $\mathfrak{F}^+$. The arithmetic operations on frequentistic states that we considered above can now be made more precise. If c and d are frequentistic states, then $c + d$ represents their sum in $\mathfrak{F}^+$. If P is a predicate, let $\chi(P)$ denote its characteristic subset of $\mathfrak{D}$. The restriction of c to P, written $c \mid P$, denotes the measure that assigns to each measurable subset M of $\mathfrak{D}$ the mass

$$(c \mid P)(M) = c(M \cap \chi(P)).$$

We must guarantee that the set $\chi(P)$ is measurable, but this is not a severe restriction. Note that the subset of $\mathfrak{F}$ consisting of those measures all of whose mass lies in $\chi(P)$ is a subspace of $\mathfrak{F}$, and the operation of restriction can be viewed as projection onto this subspace.

A program should be interpreted as a mapping from $\mathfrak{F}^+$ to $\mathfrak{F}^+$, that is, from input frequentistic state to output frequentistic state. It turns out that the mappings that interpret programs extend uniquely to continuous linear mappings from $\mathfrak{F}$ to $\mathfrak{F}$. Since the notion of a continuous linear map between vector spaces is so familiar, it is better to let these extensions define the actual meanings of programs. Thus, we shall follow Kozen and adopt the convention that the formal interpretation of a program is a continuous linear map from $\mathfrak{F}$ to $\mathfrak{F}$, where $\mathfrak{F}$ is the space of all measures on $\mathfrak{D}$. Kozen proves that these maps will have two properties. First, they will take positive measures into positive measures, as we would expect. Secondly, the total amount of mass that comes out of any program will never exceed the total amount of mass that went in. We can state this formally by the inequality

$$\|f(c)\| \leq \|c\| \tag{2.1}$$

where $f\colon \mathfrak{F} \to \mathfrak{F}$ denotes the interpretation of a program, and c is any frequentistic state.

To complete our blitz through Kozen's results, we need to describe the manner in which the linear mappings that interpret programs are built up. These rules are the essence of Kozen's semantics, since they give a formal meaning to each of the constructs out of which probabilistic **while**-programs are built. We will consider these language constructs in turn.

The empty statement is interpreted as the identity map from $\mathfrak{F}$ to $\mathfrak{F}$.

Next, consider the assignment statement $X \leftarrow e$, where $X = X_i$ is the ith program variable and $e = e(X_1, \ldots, X_n)$ is an expression in the program variables. The deterministic effect of this statement is described by the function v from $\mathfrak{D}$ to $\mathfrak{D}$ that takes the deterministic state

$$\langle x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n \rangle$$

to the state

$$\langle x_1, \ldots, x_{i-1}, e(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n \rangle.$$

Kozen then interprets the frequentistic effect of the statement $X \leftarrow e$ as the linear mapping $f\colon \mathfrak{F} \to \mathfrak{F}$ that takes the input measure c to the output measure $c \circ v^{-1}$. The symbol "$\circ$" denotes functional composition, and we define the set mapping v^{-1} by the usual rule,

$$v^{-1}(M) = \{m \mid v(m) \in M\}.$$

The statement that performs a random choice is written rather like an assignment. If F denotes a probability distribution (positive measure of norm 1) on the data type D_i of X_i, then we can choose a value for X_i at random from the distribution F by executing the statement $X_i \leftarrow \mathrm{Random}_F$. This statement is interpreted as the linear mapping $f\colon \mathfrak{F} \to \mathfrak{F}$ that satisfies the identity

$$\bigl(f(c)\bigr)\bigl(M_1 \times \cdots \times M_n\bigr) = c(M_1 \times \cdots \times M_{i-1} \times D_i \times M_{i+1} \times \cdots \times M_n)F(M_i);$$

here, each M_j is an arbitrary measurable subset of the corresponding data type D_j, and c is any measure in $\mathfrak{F}$. Since our σ-algebra for $\mathfrak{D}$ is generated by the rectangles, this identity is enough to define the measure $f(c)$.

The interpretations of larger programs are constructed recursively as follows. If programs S and T have interpretations f and g respectively, then the program "$S; T$" has as interpretation the composed function $g \circ f$. The conditional statement

if P **then** S **else** T **fi**

is interpreted as the function $h\colon \mathfrak{F} \to \mathfrak{F}$ defined by

$$h\colon c \mapsto f(c \mid P) + g(c \mid \neg P).$$

Kozen also allows loops in his programs; even though we haven't progressed that far ourselves, we shall discuss his coverage here. Consider the looping construct

while P **do** S **od.**

The semantic interpretation of this **while**-loop has to be some continuous linear mapping from $\mathfrak{F}$ to $\mathfrak{F}$; call the space of all such mappings $\mathfrak{L}$. We want the semantics of the **while**-loop to be identical to the semantics of the composite construct

if P **then** S; **while** P **do** S **od else nothing fi.**

This means that the linear map associated with the **while**-loop must be a fixed point of the affine transformation $\tau\colon \mathfrak{L} \to \mathfrak{L}$ that maps a linear transformation $h\colon \mathfrak{F} \to \mathfrak{F}$ in $\mathfrak{L}$ to the linear transformation $\tau(h)$ defined by

$$\tau(h)\colon c \mapsto h\bigl(f(c \mid P)\bigr) + (c \mid \neg P);$$

the function f once again represents the interpretation of the program S. Furthermore, if we want our semantics to agree with the normal model of computation, we want to choose the least fixed point of τ. The affine mapping τ will have multiple fixed points when a process can get stuck in the loop forever with nonzero probability; the non-least fixed points assign some result value to the program in these nonterminating cases.

That is enough of an introduction to Kozen's semantics for the time being. In Chapter 4, we will present the rules of the frequency system, and prove their soundness with repsect to Kozen's semantics. At that point, we will have to recall the above rules once again. For now, it is enough to know that there is a reasonable semantics for **while**-programs that interprets each program as a linear mapping between vector spaces of measures, and hence backs up the intuitive chromatic plumbing metaphor. Kozen also shows that this linear mapping semantics is equivalent to a more straightforward semantics based upon functions from random variables to random variables.

The Arithmetic Connectives.

We return to the question of writing probabilistic assertions that describe the behavior of loop-free programs. The function of these assertions is essentially to determine a certain subset of $\mathfrak{F}^{+}$, the set of all frequentistic states, or equivalently, of all demon reports. Our current position is that an assertion should be built out of atomic assertions of the form

$$\Pr(P) = e \qquad \text{or} \qquad \mathrm{Fr}(P) = e.$$

The probabilistic atomic assertion $\Pr(P) = e$ holds for precisely those measures c in $\mathfrak{F}^{+}$ that satisfy the relation

$$\frac{c\bigl(\chi(P)\bigr)}{c(\mathfrak{I})} = e,$$

where $\chi(P)$ denotes P's characteristic subset of $\mathfrak{I}$; in words, the real value e between 0 and 1 gives the percentage of the mass of c that satisfies P. The frequentistic atomic assertion $\mathrm{Fr}(P) = e$ is a little simpler; it holds for precisely those measures c in $\mathfrak{F}^{+}$ that satisfy the relation

$$c\bigl(\chi(P)\bigr) = e,$$

that is, for those measures which ascribe mass e to the characteristic set of P.

These atomic assertions will be combined with connectives. One possible family of connectives is the logical connectives "and", "or", and "not" and the quantifiers "for all" and "there exists". These connectives correspond to performing the elementary set-theoretic operations on the associated characteristic sets. We now have enough understanding to define a new collection of connectives, which correspond to the arithmetic operations on the characteristic sets.

Let A and B denote assertions, and hence also subsets of $\mathfrak{F}^{+}$. The first arithmetic connective is addition; the assertion $A+B$ denotes the set of all positive measures that can be expressed as the sum of a measure in A and a measure in B. Similarly, we can define a restriction operation on assertions, which is a generalization to sets of the restriction operation on frequentistic states; in particular, if P denotes a predicate, the assertion $A \mid P$ denotes the set of all measures of the form $a \mid P$ for some a in A.

The importance of these arithmetic connectives can be seen by a comparison with the Floyd-Hoare situation. In Floyd-Hoare verification, a predicate describes a certain subset of $\mathfrak{I}$, the set of all deterministic states. And in a Floyd-Hoare system, the connectives that are needed to describe the actions of forks and joins are the logical ones. If the predicates P and Q describe the two in-branches of a join, then the predicate $P \vee Q$ describes the out-branch of the join. If the predicate P describes the in-branch of a test of B, then $P \wedge B$ describes the TRUE out-branch and $P \wedge \neg B$ describes the FALSE out-branch. In our probabilistic world, however, an assertion describes a certain subset of $\mathfrak{F}^{+}$, the set of all positive measures, and the arithmetic connectives are the ones that describe the actions of forks and joins. If the assertions A and B describe the two in-branches of a join, then the assertion $A+B$ describes the out-branch. If the assertion A describes the in-branch of a test of P, then $A \mid P$ describes the TRUE out-branch and $A \mid \neg P$ describes the FALSE out-branch.

With this understanding of the probabilistic world, we can begin to get some sense of what the rules of a formal system for algorithmic analysis will be like. In particular, the rules that deal with control structure can be found by using the above guidelines to reflect the flowchart structure that lies behind each of the syntactic control structures of the language. For example, if we use square brackets rather than braces to distinguish an assertion from a predicate, the Rule for the Conditional Statement will be:

$$\frac{\vdash [A \mid P]\, S\, [B], \quad \vdash [A \mid \neg P]\, T\, [C]}{\vdash [A]\ \textbf{if}\ P\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}\ [B+C]}.$$

This rule is sound because it corresponds to Kozen's semantics for conditionals. But before we follow these ideas further, it is high time that we allowed loops in our programs once again.

Chapter 3. Living with Loops

Loops in Plumbing Networks.

Loops in a chromatic plumbing network aren't much of a problem. In fact, when we first considered the deterministic version of the chromatic plumbing metaphor, we allowed loops. In a network with loops, the control pellet might pass the same point on the network several times, in the same or in different states. If the modeled computation does not halt, then the control pellet will spend eternity going around and around the loops, changing color as appropriate to model the non-terminating computation. We can extend the probabilistic version of the chromatic plumbing metaphor to include loops by the same technique. Each weighted pellet travels around the network independently, possibly passing the same point many times. A particular pellet will emerge from an output chute if and only if the computation that it is modeling halts.

A demon on the network still reports the total masses of pellets of each color that have gone by. In particular, the demon has no sense of time passing, and does not distinguish between pellets that go by early in the computation from those that go by later on; the demon only reports total weights. Recall that in loop-free programs, if we normalized the input mass to have a total weight of 1 gram, then the weight reported by a demon for the color yellow was exactly the probability that control would pass that point in the flowchart in the yellow state during a random execution of the program. Now that we are allowing loops, the weight reported by a demon for yellow is simply the expected total weight of yellow pellets that pass that point during a random execution.

The presence of loops does change the character of a demon's report somewhat, however: the demon may report an infinite amount of mass. In fact, there may be an infinite amount of mass of a particular color, or there may merely be an infinite amount of mass all told, although each color's total remains finite. Thus, in the looping world, a demon's report is no longer guaranteed to be an element of $\mathfrak{F}^+$, the set of all positive measures on deterministic states. Instead, the report is a possibly infinite measure.

Let **R** denote the real numbers, and let **R*** denote the nonnegative real numbers with the special element "∞" representing infinity added. We can do arithmetic in **R***, although we have to be careful about such indeterminate expressions as $\infty - \infty$. It is possible to define measures that have **R*** as their range rather than **R**; in fact, **R*** has some advantages over **R** in measure theoretic contexts, since every increasing sequence has a least upper bound in **R***, either finite or infinite. We shall define $\mathfrak{F}^*$ to be the set of all countably additive **R***-valued set functions on the measurable space $\mathfrak{D}$ of all deterministic states; that is, an element of $\mathfrak{F}^*$ is a positive, possibly infinite measure on $\mathfrak{D}$. Note that neither $\mathfrak{F}$ nor $\mathfrak{F}^*$ contains the other, although they both contain $\mathfrak{F}^+$. A demon's report in the domain of programs with loops is simply an element of $\mathfrak{F}^*$.

Probabilistic Assertions and Loops.

Our next task is to determine the effects of loops upon the structure of our assertions about probabilistic state. We shall consider as our motivating example the algorithm FindMax for finding the maximum of a random permutation by a left-to-right scan. The analysis of FindMax served as our paradigmatic example in Chapter 1 of the average case analysis of a deterministic algorithm. Recall that the program is

```
$M \leftarrow X[1]$;
for $J$ from 2 to $N$ do
    if $X[J] > M$ then $M \leftarrow X[J]$ fi od.
```

We shall be guided in designing assertions to go on loops by the way that we have handled loops in the chromatic plumbing metaphor. On a loop, we shall make an assertion—something like a loop invariant in Floyd-Hoare verification—that describes all at once everything that happens around the loop. In particular, the probabilistic assertions that we put on the loop will describe the demon reports that come back from the loop in the plumbing network. We shall indicate points on the network with Greek letters in the program, enclosed in double brackets. In FindMax, we might associate a loop-descriptive assertion either with the beginning of the loop body at the point β, or with the end at the point γ:

```
$M \leftarrow X[1]$ $[\![\alpha]\!]$;
for $J$ from 2 to $N$ do
    $[\![\beta]\!]$ if $X[J] > M$ then $M \leftarrow X[J]$ fi $[\![\gamma]\!]$ od.
```

The analysis of the FindMax program will have to include a description of the probabilistic distribution of the current maximum M. This kind of information will presumably come either from atomic assertions of the form $\Pr(M = m) = c_m$ or of the form $\mathrm{Fr}(M = m) = c_m$. For example, just after the assignment $M \leftarrow X[1]$, at control point α, we could describe the distribution of M either by the probabilistic assertion

$$\Pr(M = X[1]) = 1$$

or by the frequentistic assertion

$$[\mathrm{Fr}(M = X[1]) = 1] \wedge [\mathrm{Fr}(M \neq X[1]) = 0].$$

But now consider what an assertion on the loop might be like. From a mathematical point of view, we would like to consider M to be a different random variable each time through the loop; in particular, at γ, the end of the loop body, M will be equal to the maximum of the first J elements of the input array, and will have the distribution of that maximum. Therefore, we want to describe the distribution of M as a function of J. If we are using frequentistic atomic assertions, this isn't difficult. All we have to do is assert the conjunction of a class of atomic assertions of the form

$$\mathrm{Fr}([M = m] \wedge [J = j]) = c_{m,j}.$$

This type of assertion can describe the distribution of M for each possible value of J completely independently. In fact, these assertions really give the joint frequency distribution of M and J, and hence treat M and J symmetrically.

On the other hand, suppose that our atomic assertions were of the probabilistic variety, that is, of the form $\Pr(P) = e$. In the loop-free case, we could define what this type of statement meant in terms of the reports of demons in the chromatic plumbing metaphor; in particular, we defined the expression $\Pr(P)$ to denote the fraction by weight of all of the pellets reported that satisfied the predicate P, or equivalently,

$$\Pr(P) = \frac{\mathrm{Fr}(P)}{\mathrm{Fr}(\textsc{true})}.$$

In the presence of loops, it is much harder to decide upon an appropriate denominator for this ratio. There are several decisions we could make on this question, but none of them are completely satisfactory. We want to determine some partition of the set of all pellets that pass by a demon, and then do our probability calculations on each class of this partition separately. Dividing by the total weight of the class will scale things so that each class looks like a probability space by itself.

The first option is to put all of the pellets into one big class, which means that we continue to divide by the total weight of all of the pellets that pass the demon. In our example, however, note that this total weight is larger than one gram. In fact, the total weight of all pellets passing points β and γ in the network will be precisely $N-1$ grams, since the body of the loop is always executed precisely $N-1$ times. Therefore, if we choose this option, probabilistic assertions at either β or γ will behave like frequency assertions that have been rescaled by a factor of $N-1$. That might be helpful if the same thing were happening each time around the loop; dividing by the total mass would then just remove the multiplicative factor of $N-1$ from the frequencies. But we just agreed that we would like to describe the distribution of M independently for each possible value of J. Thus, the rescaling would just be a nuisance. We don't want to treat all of the pellets at once when we assert our probabilities, but rather only those pellets that correspond to "this time around the loop."

This suggests a second alternative, which is the option that Ben Wegbreit adopted in the construction of his formal system [33]. Note that, in this example program at least, we intuitively want to consider M as a random variable but J as a non-random variable. The behavior of J can be analyzed by Floyd-Hoare techniques, since there is nothing random about it. The random behavior is centered in the input array X and the variable M. When such a clear distinction exists between random and non-random components of the process state, we can choose to treat those components differently; in particular, we can partition the set of all pellets on the basis of the values of the non-random variables. In FindMax, we would then make atomic assertions of the form

$$\Pr(M = m) = c_{m,J},$$

where this would be interpreted in terms of the demon reports as

$$\bigwedge_j \left[\frac{\mathrm{Fr}([M = m] \wedge [J = j])}{\mathrm{Fr}(J = j)} = c_{m,j} \right].$$

That is to say, non-random variables would be treated as in Floyd-Hoare systems, and could hence appear on the right-hand side of probabilistic atomic assertions. Such an assertion is interpreted as describing the proportion by weight of those pellets corresponding to a particular combination of values of the non-random variables that also satisfy the stated predicate. Note that, in the case of FindMax, this idea of partitioning the process state into random and non-random coponents works very well, and allows us to give the distribution of M as a function of J just as we desire.

The program InsertionSort, which is the major example in Ben Wegbreit's paper, also has the property that the process state can be cleanly partitioned into random and non-random components; the array being sorted is random, while the pointers into that array are non-random. Despite the success of these two examples, however, it is by no means clear that this partitioning of the process state will be easy or even feasible in general. It might be the case that all of the program variables display random behavior of one sort or another. Even if a partition is possible, it is a little unpleasant to have to treat non-random variables differently from random ones; it would be simpler if a non-random variable were simply a random variable whose distributions all had their mass concentrated at a single point.

There is a third possibility that is worth mentioning just to demonstrate its problems. We could partition the set of all pellets into subsets by considering the execution history of each pellet. Those pellets that had taken precisely the same path through the plumbing network from the input funnel to their current location would be deemed to be in the same class, and the quantity $\Pr(P)$ would denote the percentage by weight of the pellets in such a class that satisfied predicate P. It is best to think about this scheme in terms of the chromatic plumbing network. Suppose that instead of dumping a bag of separate pellets into the input funnel, we instead dropped a pie-chart, whose slices were sized and colored to model the input distribution. At a fork in the network, a pie-chart would break into two smaller pies, describing the TRUE and FALSE slices of the input pie respectively. At an assignment box, the pie slices would be recolored as dictated by the assignment, and, at a join, any pie coming in either in-branch would proceed independently down the out-branch. Then, if we view a probabilistic atomic assertion as describing the sizes of pie slices, say as fractions of 2π radians, we have a model for this execution history scheme.

At first blush, such a pie-slicing scheme looks pretty good. Without any partitioning of the process state into random and non-random components, it manages to partition the set of all pellets in a reasonable way. In a simple **for**-loop, for example, we would expect this scheme to put into one class all those pellets that had gone around the loop the same number of times. But think about our FindMax example. The body of the loop is itself an **if**-statement, and every time that the pies pass through this **if**-statement, they will be further divided. Thus, we shall be left computing probabilities over too fine a partition; two pellets will be equivalent only if (i) they have gone around the **for**-loop the same number of times, and (ii) they have followed the same branch of the **if**-test each time around.

The problem of overly fine partitions shows up in other funny ways as well. For example, in a formal system based on a pie-slicing scheme, the program

$$\textbf{if } K = 0 \textbf{ then nothing else nothing fi}$$

is not a no-op. Suppose that a pie with two slices of equal size, one colored $K = 0$ and the other colored $K = 1$ enters the input funnel; that would correspond to the input assertion

$$[\Pr(K = 0) = \tfrac{1}{2}] \wedge [\Pr(K = 1) = \tfrac{1}{2}].$$

Then, there will emerge from the output chute two distinct pies, one colored $K = 0$ and the other colored $K = 1$. The input assertion would not correctly describe this output state; instead, we would have to make the output assertion

$$[\Pr(K = 0) = 1] \vee [\Pr(K = 1) = 1],$$

where each atomic assertion describes one of the pies. And it is very unfortunate to have a program that does nothing, but still affects the assertions that move through it. Thus, although partitioning on execution history tends to divide up the set of all pellets in something like the right way, it isn't right enough in general to build into a formal system. If we had some way to specify which characteristics of the execution history should cause pies to split and which should not, a scheme based on pie-slicing might work very nicely.

The net result of all this is that there doesn't seem to be any good way to partition up the pellets for scaling purposes. The unpleasant characteristics of any of the above schemes seem to outweigh the relatively minor hassles of using frequencies throughout. And in addition, the use of frequencies solves the Leapfrog problem. The only way in which frequencies are less pleasant to work with than probabilities is that frequencies don't necessarily sum to 1. But the only way to guarantee that the right-hand sides continue to sum to 1 is to perform some sort of rescaling, and all choices for these rescalings run into difficulties of one sort or another.

Therefore, we shall hereby give up on probabilities entirely. In the future, we shall stick to atomic assertions that talk about frequencies instead.

The presence of loops does present one challenge even to those who have been converted to a frequentistic way of thinking, however: what about infinite mass? As we pointed out earlier, the presence of loops implies that the reports of demons will not necessarily lie in $\mathcal{F}^+$, although they will lie in $\mathcal{F}^*$. So far, we have been considering the characteristic set of an assertion to be a certain subset of $\mathcal{F}^+$. Unless we were to change that definition, and to think instead of an assertion as describing a subset of $\mathcal{F}^*$, there is no chance that the assertions we make on loops can really describe the reports that demons on those loops will send back. We shall see shortly that there are other reasons why the assertions we put on loops won't necessarily describe the reports of demons on those loops accurately. Therefore, we shall postpone the resolution of the $\mathcal{F}^+$ versus $\mathcal{F}^*$ question until after we have explored the issues further.

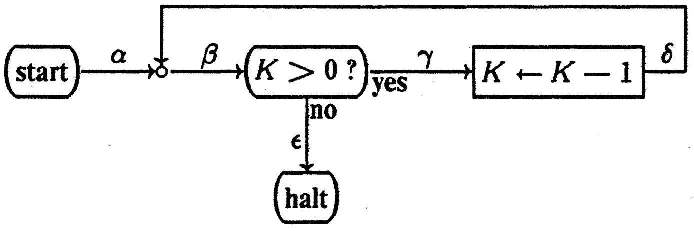

Figure 3.1. The Flowchart of CountDown.

Summary Assertions.

To further develop our knowledge and intuition about programs with loops, it is important that we do an example in some detail. In order to tackle FindMax, we have to face the thorny question of how to describe the distribution of a random permutation formally, since the input to FindMax is one of them, and random permutations are tricky. So we shall postpone a detailed treatment of FindMax for now, and consider instead a simpler example program:

$$\text{CountDown:}\quad \textbf{while } K > 0 \textbf{ do } K \leftarrow K - 1 \textbf{ od}.$$

Let n represent a fixed, nonnegative integer. We shall start off the CountDown program with one gram of execution mass in which $K = n$ with certainty; that is, we shall assume the input assertion

$$[\mathrm{Fr}(K = n \geq 0) = 1] \wedge [\mathrm{Fr}(K \neq n) = 0].$$

This assertion is associated with the control point α, where the control points α through ϵ are shown in Figure 3.1 and indicated textually below:

$$[\![\alpha]\!] \textbf{ while } [\![\beta]\!]\ K > 0 \textbf{ do } [\![\gamma]\!]\ K \leftarrow K - 1\ [\![\delta]\!] \textbf{ od } [\![\epsilon]\!].$$

A glance at the flowchart already shows us some things about what demons will report; for example, the report from point β will be exactly the sum of the reports from points α and δ.

What assertions shall we make at the various control points? First, we shall let ourselves be guided by our intuition of what really happens, and attempt to characterize that truth with our assertions. This approach leads us to the assertions

$$\begin{aligned}
&\alpha\text{: } [\mathrm{Fr}(K \neq n) = 0] \wedge [\mathrm{Fr}(K = n \geq 0) = 1]\\
&\beta\text{: } [\mathrm{Fr}([K < 0] \vee [K > n]) = 0] \wedge \bigwedge_{0 \leq k \leq n} [\mathrm{Fr}(K = k) = 1]\\
&\gamma\text{: } [\mathrm{Fr}([K \leq 0] \vee [K > n]) = 0] \wedge \bigwedge_{0 < k \leq n} [\mathrm{Fr}(K = k) = 1]\\
&\delta\text{: } [\mathrm{Fr}([K < 0] \vee [K \geq n]) = 0] \wedge \bigwedge_{0 \leq k < n} [\mathrm{Fr}(K = k) = 1]\\
&\epsilon\text{: } [\mathrm{Fr}(K \neq 0) = 0] \wedge [\mathrm{Fr}(K = 0) = 1].
\end{aligned} \tag{3.1}$$

If K is the only program variable, the assertions (3.1) all specify a frequentistic state exactly; that is, they each have a characteristic set that contains precisely one point of $\mathcal{F}^+$. If there are other program variables besides K, and hence other components in the process state, then each of these assertions describes some larger subset of $\mathcal{F}^+$. The important thing to note, however, is that in either case these assertions look everywhere locally correct. That is, an individual looking at the input and output assertions of any single flowchart feature by itself would agree that the output assertions follow from the input assertions. In some cases, this consistency just reflects an arithmetic fact about the subsets of $\mathcal{F}^+$ that these assertions describe; we have

$$\begin{aligned}\beta &= \alpha + \delta\\ \gamma &= \beta \mid (K > 0)\\ \epsilon &= \beta \mid (K \leq 0)\end{aligned}$$

where each Greek letter in these equations stands for the corresponding assertion in (3.1). There should be one more equation relating our five assertions, since we expect the input assertion to determine the rest. This remaining relation concerns the affect of the assignment statement, and this is something that we haven't yet considered in detail. But, when we do, it seems clear that we shall agree that the truth of γ before the assignment $K \leftarrow K-1$ implies the truth of δ thereafter.

Our hope is to build a formal system in which the correctness of an augmented program can be verified by just checking that the program is everywhere locally correct in the above sense. In particular, the CountDown program augmented with the assertions of (3.1) should be a theorem of the system. This suggestion is the probabilistic analog of what happens in Floyd-Hoare verification.

In Floyd-Hoare systems, the difficult characteristic of a loop is the fact that execution mass keeps coming back to the top of the loop and joining the input arbitrarily often. When the mass coming around the loop joins the input stream, the logical operation performed in Floyd-Hoare is an "or". Therefore, a predicate on the loop must be the "or" over all n of the predicates that would describe the mass going around the loop for the nth time. A loop-cutting predicate in a Floyd-Hoare system, then, is an *invariant*, the result of an infinite disjunction in some sense. In our probabilistic world, it is still mass coming back to the top of the loop arbitrarily often that is the problem, but the connective that occurs at that join is "plus" rather than "or". Therefore, the loop cutting assertions in our system are the limits of infinite summations; we shall call them *summary assertions* to contrast them with the invariants of Floyd-Hoare. The summary assertion of a loop describes all of the mass that will ever go around the loop.

It is convenient to associate with each looping construct in the language a particular point in the corresponding flowchart at which to make *the* summary assertion for that construct. This choice is basically arbitrary, but it is good to establish an explicit convention at the outset, and stick to it. For example, consider the **while**-loop of the CountDown program; we might conceivably pick any of the three control points β, γ, or δ, and distinguish the corresponding assertion as the summary assertion for the loop. Point β would correspond to the convention, "Describe all of the mass about to enter the control condition"; point γ would be, "Describe the mass about to begin execution of the loop body"; and point δ would be, "Describe the mass emerging from the loop body". It would be hard to choose between γ and δ, so we shall adopt the β alternative as our convention: a summary assertion describes the execution flow at the point where it enters the control test, the test that will determine whether or not the loop exits. Note that this alternative makes our summary assertions just a little more "summary" than either of the others; every pellet goes past the point β one more time than it goes through the loop body itself. This convention also generalizes nicely to handle the more general looping construct

loop S **while** $[\![\varsigma]\!]$ P: T **repeat;**

we shall stipulate that the summary assertion describes the flow through the point ς.

A further convention needs to be established for **for**-loops, dealing with the loop index. Consider the **for**-loop

for J **from** ℓ **to** u **do** S **od.**

To be consistent with our convention for **while**-loops, we presumably want the summary assertion of the **for**-loop to describe all of the flow entering the test of the loop index J against the upper bound u. But we must decide whether the assertion will use the incremented or non-incremented value of J. For example, suppose that the explicit **for**-loop

for J **from** 1 **to** n **do nothing od**

is entered with one gram of mass, so that $\mathrm{Fr}(\text{TRUE}) = 1$ on input. A convention employing the incremented value of J would dictate the summary assertion

$$\big[\mathrm{Fr}([J < 1] \lor [J > n + 1]) = 0\big] \land \bigwedge_{1 \le j \le n+1} \big[\mathrm{Fr}(J = j) = 1\big],$$

but a convention employing the non-incremented value of J would give

$$\big[\mathrm{Fr}([J < 0] \lor [J > n]) = 0\big] \land \bigwedge_{0 \le j \le n} \big[\mathrm{Fr}(J = j) = 1\big]$$

as the summary assertion. The difficulty stems from the fact that, while the body of the loop is executed exactly n times, corresponding to the n integers between 1 and n, the summary assertion must describe $n + 1$ grams of mass. These two conventions describe this extra mass as the $J = n + 1$ mass and the $J = 0$ mass respectively.

The latter convention actually works out more neatly from a notational point of view. But unfortunately, it is hard to convince a programmer that a **for**-loop from 1 to n actually starts at 0. The standard implementation of the **for**-loop in terms of a **while**-loop is

```
$J \leftarrow \ell$;
while $J \le u$ do $S$; $J \leftarrow J + 1$ od.
```

To allow us to think in terms of this standard implementation, we shall adopt the former convention, in which the summary assertion of a **for**-loop is just the summary assertion of the **while**-loop in its standard implementation. We shall live with the notational inconvenience that this convention generates.

It is also convenient to choose some point in the program text where the summary assertion can be said to hold. When a **for**-loop is expanded out as a **while**-loop, this is no problem; the control point of the summary assertion is just before the $J \leq u$ test. But when we write the loop as a **for**-loop, there really isn't any good place. We shall somewhat arbitrarily choose to put it right before the **do**, at the point labelled σ in

$$\textbf{for } J \textbf{ from } \ell \textbf{ to } u \ [\![\sigma]\!] \textbf{ do } S \textbf{ od}.$$

Parenthetically, note that our example assertions only make sense if n is nonnegative, so that the loop is executed at least zero times. It is generally the case that **for**-loops with $u < \ell - 1$ cause more trouble to verification efforts than they are worth; we hereby forbid them.

Fictitious Mass.

In the last section, we saw that the assertions (3.1) that describe the actual behavior of the CountDown program have the property that they look everywhere locally correct. Reasoning in that direction is the easy part. The more interesting question is the converse: if a group of assertions look everywhere locally correct, does this mean that they do in fact describe reality? As one might expect, funny things can happen when one attempts to reason in this direction.

For a first example, consider the completely trivial looping program

$$\text{EmptyLoop:} \ \ \textbf{while } K > 0 \textbf{ do nothing od;}$$

and assume that this program is started in a frequentistic state satisfying the assertion

$$[\mathrm{Fr}(K = 0) = 1] \wedge [\mathrm{Fr}(K \neq 0) = 0]. \tag{3.2}$$

From our knowledge of programming reality, we can see that the body of the loop in EmptyLoop will never be executed; the one gram of mass that enters the input funnel will fail the $K > 0$ test, and will fall out after zero iterations of the loop body. We can describe this reality by using the input assertion as the summary assertion for the loop. On the other hand, consider the following assertion as a candidate for a summary assertion:

$$[\mathrm{Fr}(K = 0) = 1] \wedge [\mathrm{Fr}(K = 4) = 7] \wedge [\mathrm{Fr}([K \neq 0] \wedge [K \neq 4]) = 0]. \tag{3.3}$$

This summary assertion not only describes the one gram of mass with $K = 0$ that we discussed above, it also claims the existence of seven grams of mass in which K has the value 4. Of course, these seven grams don't correspond to anything that happens in the real world. But let us consider how this summary assertion looks from an everywhere local point of view. The summary assertion, of course, describes all of the mass entering the control test. Note that the one gram with $K = 0$ will be rejected by this test, and will leave the loop as expected; the seven grams with $K = 4$, however, pass the test, and enter the body of the loop. Since the body of the loop is empty, these seven grams emerge unscathed at the end of the loop body, and now they can combine with the one gram of input mass to support all of the mass described by the summary assertion.

What is going on here? The loop of our trivial program has the property that pellets of some colors will travel around the loop completely unchanged. If we choose a summary assertion that ascribes nonzero mass to any such color of pellet, the assertion will look everywhere locally correct, even though it is global nonsense. We shall call this phenomenon *fictitious mass*; that is, our example assertion describes seven fictitious grams of execution mass going around the loop, the seven in which $K = 4$.

There is no obvious way to eliminate the possibility of fictitious mass. In any particular case, we can prove that the fictitious mass doesn't really happen. In the EmptyLoop case, for example, we can argue by induction that, since K is never 4 on entry to the loop, and since the loop can't produce $K = 4$ mass out of other mass, there won't ever be any $K = 4$ mass going around the loop. In fact, we noted earlier that the input assertion (3.2) is an acceptable summary assertion for the loop, and this proves that the $K = 4$ mass must be fictitious. But that doesn't eliminate the problem that the summary assertion (3.3) also looks everywhere locally correct.

Note that the fictitious mass described by summary assertion (3.3) is caught entirely inside the loop, however. If we perform an anlysis of EmptyLoop with assertion (3.3) as the summary for the loop, we would deduce that the output assertion should be

$$[\mathrm{Fr}(K = 0) = 1] \wedge [\mathrm{Fr}(K \neq 0) = 0],$$

the same as the input assertion and in fact, the correct result. Although the summary assertion is describing more than what really happens, the extra stuff, the fictitious mass, is confined to the loop, and has no effects that are visible from outside the loop.

In Kozen's semantics, a loop is interpreted as the least fixed point of an affine transformation. It might appear at first glance that this definition eliminates the problem of fictitious mass, but in fact there is really no connection between fictitious mass and the non-least fixed points of the affine transformation. A non-least fixed point assigns a value to the program in the cases where it would "really" run forever; and the effect of this is visible from outside the loop. Fictitious mass, on the other hand, is not visible from outside the loop; in fact, it only makes sense to talk about fictitious mass if you are describing what goes on inside the loop.

Fictitious mass comes in more interesting flavors as well. First, suppose that a loop has the property that one gram of red mass is transformed into one gram of green mass by going around the loop once, and vice versa. Then, a summary assertion for such a loop can assert the presence of an arbitrary amount of red and green mass, as long as the amounts are equal:

the red will support the green and the green will support the red. Similarly, fictitious mass can involve a cycle of states of any length. We can also get fictitious mass from an infinite sequence of states in a chain, instead of from a finite sequence of states in a cycle. If the program

$$\mathbf{while}\ K > 0\ \mathbf{do}\ J \leftarrow J + 1\ \mathbf{od}$$

is executed by a process starting with one gram of mass in which K is 0, the summary assertion that reflects reality is

$$[\mathrm{Fr}(K = 0) = 1] \wedge [\mathrm{Fr}(K \neq 0) = 0].$$

But the summary assertion

$$[\mathrm{Fr}(K = 0) = 1] \wedge [\mathrm{Fr}([K \neq 0] \wedge [K \neq 3]) = 0] \wedge \bigwedge_j [\mathrm{Fr}([K = 3] \wedge [J = j]) = 5]$$

also looks everywhere locally correct, even though it describes in some sense five fictitious executions of the loop in which K is 3 and J counts through all the integers.

No matter what the flavor of fictitious mass, it is still the case that all of the effects of that mass are confined to the inside of the loop around which the mass is circulating. Unfortunately, this confinement is not the case with time bombs.

Time Bombs.

Consider again the CountDown example program

$$\mathbf{while}\ K > 0\ \mathbf{do}\ K \leftarrow K - 1\ \mathbf{od},$$

starting it off this time with a frequentistic state satisfying

$$[\mathrm{Fr}(K = 0) = 1] \wedge [\mathrm{Fr}(K \neq 0) = 0]. \tag{3.4}$$

What will really happen is precisely nothing; the one gram of input mass will fail the control test $K > 0$ and exit immediately. But suppose that we decide instead to try out the summary assertion

$$[\mathrm{Fr}(K < 0) = 0] \wedge [\mathrm{Fr}(K = 0) = 8] \wedge \bigwedge_{k \geq 1} [\mathrm{Fr}(K = k) = 7]. \tag{3.5}$$

This assertion turns out to support itself around the loop. First, the mass it describes hits the control condition; the eight grams in which $K = 0$ are steered out of the loop, while all of the rest of the mass, seven grams with $K = k$ for all positive k, is steered around the loop again. The net effect of the loop body is to turn this mass into seven grams with $K = k$ for all nonnegative k; and this mass, when added to the one gram of input mass, gives us just what we need to support the summary assertion again.

This is quite a serious matter, since our analysis suggests that there is a program which control exits eight times as often as it enters! In fact, by similar reasoning, we could use the input assertion $\mathrm{Fr}(\text{TRUE}) = 0$ and the summary assertion

$$[\mathrm{Fr}(K < 0) = 0] \wedge \bigwedge_{k \geq 0} [\mathrm{Fr}(K = k) = 7],$$

to show that the program CountDown can also be exited seven times on the average even when it is not entered at all. This general phenomenon might be called a *time bomb*; there is an infinite amount of mass circulating around the loop, ticking all the while, and when some of it gets down to the $K = 0$ state it exits from the loop. Note that, while fictitious mass is merely unpleasant, the presence of time bombs is fatal to a formal system. Once a system allows the deduction of one false result, then (for most logical systems at least) every formula can be deduced, and one loses interest in the system.

We can learn to tolerate fictitious mass, but we have to get rid of time bombs somehow. Basically, to get rid of them, we shall outlaw summary assertions that describe an infinite amount of mass. It will take us a while, however, before we can flesh out this insight, and show that such a restriction really does restore the soundness of our system.

It is too bad that assertions describing an infinite amount of mass have to go. From another point of view, they would be very helpful. Consider the program

CountUp: **while** $K > 0$ **do** $K \leftarrow K + 1$ **od**.

If a process begins to execute this program in a state where K is 1 with certainty, the one gram of mass that enters the loop will be caught inside the loop forever. We can describe what really happens during the execution of this program by the input assertion

$$[\mathrm{Fr}(K = 1) = 1] \wedge [\mathrm{Fr}(K \neq 1) = 0] \tag{3.6}$$

and the summary assertion

$$[\mathrm{Fr}(K \leq 0) = 0] \wedge \bigwedge_{k \geq 1} [\mathrm{Fr}(K = k) = 1]. \tag{3.7}$$

This summary assertion does describe an infinite amount of mass, but for the excellent reason that there really is an infinite amount of mass flowing through the loop. Unfortunately, it is difficult to distinguish between legitmate situations like this one and time bombs. The question is whether the mass being described represents a realistic computation, one that started at the input funnel of the network, and has followed some finite path through the network to get to its current position. Fictitious mass represents computations that never started and will never stop, but just loop around; and time bombs represent in some sense computations that have been going around the loop since before time began, but that are just waiting to come out when the time is right. In order to distinguish between real mass, fictitious mass, and time bombs, we would have to add some notion of either time or of execution history to the chromatic plumbing metaphor. But the fact that chromatic plumbing deals only with the time-integrated flow of control through the program is one of its strong points. Instead of trying to add a notion of time, we shall concentrate on seeing how far we can get without such a notion. And, without such a notion, any dealings with infinite mass raise the specter of time bombs.

Even if we are forced to outlaw any assertions that describe an infinite amount of mass, note that we can still detect when infinite loops are occurring. The trick is to avoid describing the computations that don't terminate, and to deduce their presence from the external description of the loop, in particular, from the fact that more mass enters the loop than leaves it. For example, consider the program CountUp once again, started in a state described by assertion (3.6). We can substitute for (3.7) the less informative but still accurate summary assertion

$$[\mathrm{Fr}(K \leq 0) = 0]; \tag{3.8}$$

this assertion supports itself around the loop, and also supports the realistic output assertion $\mathrm{Fr}(\text{TRUE}) = 0$. Now, it could be argued that assertion (3.8) itself describes an infinite amount of mass; after all, the real behavior of the program, which is a point in $\mathcal{F}^*$ but not in $\mathcal{F}^+$, is an infinite measure and it satisfies (3.8). On the other hand, assertion (3.8) also describes many frequentistic states with only finite total mass including, in fact, the state with no mass at all. It will turn out that such summary assertions are legal. We can more accurately describe the assertions that must be outlawed as those that are satisfied *only* by frequentistic states with infinite total mass.

Thus, without breaking our rule about infinite mass, we can verify that no part of the one gram that enters the CountUp loop ever gets out. This demonstrates that all the parts of that gram must reflect non-terminating computations; we can deduce the presence of infinite loops even when the assertions do not explicitly describe what goes on during them. One might call this the *technique of tacit divergence*. Some of the execution mass that enters the loop may go around it infinitely often, but the summary assertion of the loop doesn't describe this mass, and its presence is instead deduced by study of the loop's input-output behavior.

The Characteristic Sets of Assertions.

Now that we have a better sense of what happens with assertions and loops, it is time to return to an issue that we left open some time ago: is the characteristic set of an assertion a subset of $\mathcal{F}^*$ or of $\mathcal{F}^+$? If we wanted the characteristic set of the summary assertion of a loop always to contain the demon's report for that loop, we would have to choose $\mathcal{F}^*$. We would also have to find some way of guaranteeing that our summary assertions did not describe any fictitious mass, and such a way does not seem to be easy to find. This suggests that there is less motivation than one might initially suspect for choosing $\mathcal{F}^*$. In addition, choosing $\mathcal{F}^*$ causes a severe difficulty in another area. If we choose $\mathcal{F}^*$, then all assertions, not just the summary assertions of loops, would describe certain subsets of $\mathcal{F}^*$. Consider what that would mean. If the input assertion of a program can describe infinite measures, then we have to define what it means to execute a program beginning with an infinite frequentistic state. This would demand a non-trivial extension of Kozen's semantics, since that semantics currently interprets a program as a linear map from $\mathcal{F}$ to $\mathcal{F}$, and hence only defines the meaning of the program for finite input measures.

This last argument is powerful enough to decide the issue. It would be pleasant if the assertion on a loop really described all of the mass that goes around that loop. And this is an excellent principle to use when devising summary assertions, as long as the technique of tacit divergence is also applied. But, from a formal point of view, we shall make the convention that an assertion describes a subset of $\mathcal{F}^+$ rather than a subset of $\mathcal{F}^*$. As a consequence, inside a loop, the report of a demon and the corresponding assertion are only tenuously related. The assertion is forbidden to describe more than a finite amount of what the demon will report, and the assertion may choose to describe some extra ficitious mass, which the demon won't report.

Chapter 4. The Frequency System

The Meaning of Theorems.

In the preceeding two chapters, we have developed some intuition for what the issues and choices are in the construction of formal systems for algorithmic analysis. In this chapter, we shall present a more precise description of a sound formal system called the *frequency system* based on these intuitions. Then, we shall turn to the study of examples of the system's use, and extensions of its power.

First, it is worthwhile to get a general sense of what the theorems in the frequency system will look like, and what they will signify about the real world. A formula in the frequency system will have the general form $[A]S[B]$, where A and B are frequentistic assertions and S is a single-entry single-exit program in a simple ALGOL-like programming language. We shall use square brackets instead of braces around our assertions to distinguish them from the predicates of a Floyd-Hoare system. The formula $[A]S[B]$ is true in a semantic sense if and only if the following statement correctly describes the chromatic plumbing metaphor: If a process begins to execute S in a (finite) frequentistic state that satisfies assertion A, then all of the execution mass that ever emerges from the normal exit of S will form a (finite) frequentistic state that satisfies assertion B. More formally, the assertions A and B describe certain subsets of $\mathfrak{F}^+$, which we shall denote $\chi(A)$ and $\chi(B)$ respectively, and the program S is interpreted as a linear map f from $\mathfrak{F}$ to $\mathfrak{F}$. The formula $[A]S[B]$ is true if and only if $f(\chi(A)) \subseteq \chi(B)$.

Note that, unlike the Floyd-Hoare partial correctness situation, there is no assumption of termination here; if A describes all of the input mass, then B will describe all of the output mass, no matter how likely or unlikely it is for S to terminate normally. Thus, the frequency system might be said to be dealing in a *strong performance logic* instead of a weak performance logic, or to be analyzing *total performance* instead of partial performance.

Despite the "strength" of its logic, the frequency system still contains Floyd-Hoare verification as a special case. Suppose that we wanted to do something in the frequency system that was equivalent to asserting in a Floyd-Hoare sense the truth of the predicate P at a certain point in a program. Floyd-Hoare systems use the method of inductive assertions; to assert predicate P at a point is to claim that, whenever control passes that point, P will hold. Or, to put it another way, P is true for all of the mass that ever passes that point. We can make precisely the same stipulation in the frequency system by asserting that $\mathrm{Fr}(\neg P) = 0$. This says that no mass ever passes the demon in which P is false, implying that P is true of whatever mass, if any, does pass the demon. Note that the atomic assertion $\mathrm{Fr}(P) = 1$ just doesn't say the correct thing at all; it specifies the total mass in which P is true, which we don't want to do, and it doesn't forbid the existence of mass in which P is false, which is the whole point. Asserting P in a Floyd-Hoare system correponds to asserting $\mathrm{Fr}(\neg P) = 0$ in the frequency system.

This one insight in fact shows us how to do Floyd-Hoare style analyses inside the frequency system. In particular, compare the Floyd-Hoare formula $\{P\}S\{Q\}$ with the formula of the frequency system that corresponds to it under the insight above,

$$[\mathrm{Fr}(\neg P) = 0]\; S\; [\mathrm{Fr}(\neg Q) = 0].$$

The Floyd-Hoare version states that, if the predicate P is true upon entry to S, and if S terminates normally, then the predicate Q will be true upon exit from S. The frequency system version states that, if the predicate P is never false upon entry to S, then the predicate Q will never be false upon exit from S. These two statements are completely equivalent, despite the fact that the first mentions termination and the second doesn't. The two negatives in the second version do not cancel each other out; they instead manage to sweep the issue of termination adroitly under the rug. This phenomenon in which a double negative finesses the assumption of termination appears also in Pratt's dynamic logic [11, 12].

There is thus a correspondence between Floyd-Hoare predicates and frequentistic atomic assertions with a zero right-hand side. We have just used this correspondence to show how to translate a Floyd-Hoare analysis into the frequency system. We can also use the correspondence in the reverse direction to help explain the structure of those frequency system analyses that we have already studied. For example, our analysis of the true performance of the CountDown program

$$[\![\alpha]\!]\ \textbf{while}\ [\![\beta]\!]\ K > 0\ \textbf{do}\ [\![\gamma]\!]\ K \leftarrow K - 1\ [\![\delta]\!]\ \textbf{od}\ [\![\epsilon]\!]$$

when started with one gram of $K = n$ mass was given by the assertions of (3.1),

$$\begin{aligned}
&\alpha\colon [\mathrm{Fr}(K \neq n) = 0] \wedge [\mathrm{Fr}(K = n \geq 0) = 1]\\
&\beta\colon [\mathrm{Fr}([K < 0] \vee [K > n]) = 0] \wedge \bigwedge_{0 \leq k \leq n} [\mathrm{Fr}(K = k) = 1]\\
&\gamma\colon [\mathrm{Fr}([K \leq 0] \vee [K > n]) = 0] \wedge \bigwedge_{0 < k \leq n} [\mathrm{Fr}(K = k) = 1]\\
&\delta\colon [\mathrm{Fr}([K < 0] \vee [K \geq n]) = 0] \wedge \bigwedge_{0 \leq k < n} [\mathrm{Fr}(K = k) = 1]\\
&\epsilon\colon [\mathrm{Fr}(K \neq 0) = 0] \wedge [\mathrm{Fr}(K = 0) = 1].
\end{aligned}$$

If we consider just the atomic assertions with zero right-hand sides, we can see that they correspond exactly to a Floyd-Hoare partial correctness analysis of the behavior of CountDown. The corresponding predicates are, after stripping off the double negatives,

$$\begin{aligned}
\alpha\colon&\ K = n\\
\beta\colon&\ 0 \leq K \leq n\\
\gamma\colon&\ 0 < K \leq n\\
\delta\colon&\ 0 \leq K < n\\
\epsilon\colon&\ K = 0.
\end{aligned}$$

This correspondence between Floyd-Hoare analysis and zero right-hand side atomic assertions is so close that we shall occasionally use it as a license to be a trifle sloppy. When we are doing a frequency system analysis of a program for which the Floyd-Hoare analysis is straightforward, we shall sometimes leave off the atomic assertions that mimic the corresponding Floyd-Hoare analysis. This convention saves a significant amount of space when writing down the analyses of more complex programs, and focuses our attention on the other atomic assertions, the ones that say something new.

Certainty versus Truth with Probability One.

This is as good a place as any to state a general caveat about the frequency system, concerning the delicate distinction between certainty and truth with probability one. When the universe that one is dealing with has an infinite amount of randomness, events might exist that are not fundamentally impossible, but that occur only with probability zero. For example, suppose that a program is given as input the results of an infinite string of independent tosses of a fair coin. The program begins to examine these tosses, looking for one that came out H, for "heads". As soon as a single H is found, the program will terminate, but as long as it continues to see only T, for "tails", it will keep looking. This program will examine precisely k of the tosses with probability 2^{-k}; the expected number of tosses examined is just 2. Yet, this program does not constitute an algorithm by the standard definition, since there is no finite bound on how long the program can run. In particular, the program will run forever with probability zero, exactly when the input happens to consist entirely of T's.

Let $\langle X[i]\rangle_{i\geq 0}$ represent the input to this program, an infinite sequence of independent random variables, where each $X[i]$ is equally likely to be H or T. In order to handle this program in the frequency system, we would need to characterize in our assertions the probabilistic structure of this input. Since the sequence $\langle X[i]\rangle$ will be independent if all of its finite initial segments are, we would want to give an assertion something like

$$\bigwedge_{\substack{n\geq 0\\ \langle x_1,x_2,\ldots,x_n\rangle\in\{\mathtt{H},\mathtt{T}\}^n}} \Big[\mathrm{Fr}\big((\forall i)(1\leq i\leq n \Rightarrow X[i]=x_i)\big)=2^{-n}\Big].$$

We can keep things simpler, however, if we allow the program to flip coins when it needs the results. Then, instead of giving the program all the randomness it will ever need in the initial input, the program can generate that randomness on the fly. For convenience in handling examples like the coin flipping program, we shall therefore allow our programs to make random choices. This is not as large a change to the framework as one might guess; remember that it was the consideration of exactly these sorts of programs that led Dexter Kozen to develop his semantics. We shall write a random assignment

$$X \leftarrow \mathbf{Random}_F,$$

where F denotes a probability distribution for values of X's data type. This assignment means that the current value of X should be replaced by a value chosen at random from the distribution F, independently of everything else that has happened. With the ability to make random choices, we can code the program CoinFlip quite neatly as a **repeat**-loop,

$$\text{CoinFlip:}\quad \textbf{loop } X \leftarrow \text{Random}_{\mathsf{HT}};\ \textbf{while } X = \mathsf{T}\ \textbf{repeat};$$

here the subscript HT represents the distribution of a fair coin, which ascribes probability $\frac{1}{2}$ to each of H and T. We shall assume that the program variable X is of data type Coin, which has only the two values H and T. With this convention, we won't have to bother carrying around the assertion

$$\text{Fr}([X \neq \mathsf{H}] \wedge [X \neq \mathsf{T}]) = 0$$

all the time; that information is built into the Coin data type.

Consider analyzing this program in the frequency system. If we put one gram of mass into the input funnel, we shall get out 2^{-k} grams after k coin flips for each positive k. All told, we shall get one gram back out again. But even though we get out one gram, we don't get out everything that we put in; mass constituting a set of measure zero is caught in the loop forever. Thus, the frequency system can deal only with total correctness from an "almost everywhere" point of view. If the same amount of mass comes out of a loop as goes in, we can conclude only that the loop terminates almost always; the user of the frequency system must be content with that assurance. In fact, this "almost everywhere" qualification occurs almost everywhere in the frequency system. We earlier described those states that satisfy the input assertion

$$\big[\text{Fr}(K = 1) = 1\big] \wedge \big[\text{Fr}(K \neq 1) = 0\big]$$

as states in which K has the value 1 with certainty; to be more accurate, we only know that K has the value 1 with probability one. Also, consider our embedding of Floyd-Hoare arguments into the frequency system. We claimed that the frequency system assertion $\text{Fr}(\neg P) = 0$ was the equivalent of the Floyd-Hoare predicate P; strictly speaking, that isn't true either. The Floyd-Hoare predicate P claims that P always holds, but the assertion $\text{Fr}(\neg P) = 0$ claims only that P is false with probability zero, not necessarily that it is always false.

In general, then, every claim that an assertion makes about the frequentistic state or behavior of a program should be qualified by a clause indicating that sets of measure zero are ignored. But now that we have discussed the situation, we shall feel free to elide this qualification most of the time.

Going back to the CoinFlip program, it is interesting to note that the summary assertion for the **repeat**-loop does not need to talk about the powers of 2 at all. Instead, we can merely assert right before the control test that

$$\big[\text{Fr}(X = \mathsf{H}) = 1\big] \wedge \big[\text{Fr}(X = \mathsf{T}) = 1\big].$$

At the control test, the one gram of H mass will exit the loop, and the one gram of T mass will remain to join the one gram of input mass. These two grams will then be evenly distributed between H and T by the random assignment, ready to support the summary assertion again.

Weak versus Strong Systems.

When we use atomic frequentistic assertions with nonzero right-hand sides, the discussion at the beginning of this Chapter indicated that we are working in a strong logic of some sort, one that has the power to talk about termination. For example, the frequency system formula

$$\big[[\mathrm{Fr}(P)=1]\wedge[\mathrm{Fr}(\neg P)=0]\big]\,S\,\big[[\mathrm{Fr}(Q)=1]\wedge[\mathrm{Fr}(\neg Q)=0]\big]$$

claims that the program S is totally correct with respect to the assertions P and Q (if we ignore sets of measure zero). It is helpful to compare the reasoning about termination that occurs in the frequency system with the methods that are normally used in total correctness program verification systems.

There are two primary methods for giving proofs of termination: counter variables, and well-founded sets. We have already discussed the method of counter variables as a way of formalizing the upper bound part of worst case analyses [exercise 1.2.1–13 in 18, 24]. In this method, a new variable is added to the program, initialized to zero, and incremented once each time control goes through the loop. Then, using standard partial correctness techniques, the value of this counter is bounded by some function of the program's input. The existence of such a bound guarantees the termination of the loop. The method of counter variables is somewhat limited for proving termination, because it requires the existence of suitable bounds on the running time of the program; it is often possible to prove the termination of a program without knowing such explicit bounds, using the method of well-founded sets [8]. In this method, a loop is shown to terminate by demonstrating that a certain expression in the program variables has values that lie in a well-founded set, and that every execution of the loop causes the value of this expression actually to decrease. Since there are no infinite decreasing sequences in well-founded sets, this proves that the loop must terminate. This method has greater applicability; that is to say, it allows proofs of termination with less explicit knowledge about the behavior of the program than the method of counter variables.

Implicit in the previous paragraph is the sense that a method for proving termination is powerful in as much as it allows proofs of correctness while specifying as little as possible about the structure of the terminating computation. In this hierarchy, the frequency system ranks as a weakling, since it can only demonstrate termination of a program (with probability one) when we are willing to describe in detail everything that ever happens during that terminating computation. As an example, consider the CountDown program once again,

$$\textbf{while } K>0 \textbf{ do } K\leftarrow K-1 \textbf{ od.}$$

Note that, regardless of the characteristics of the input distribution, this program always halts. Proving this would be a triviality for a total correctness verification system; since every execution of the loop body reduces K by 1, that body cannot be executed more times than the value of K on input. If we work from the input assertion $\big[\mathrm{Fr}(K=n)=1\big]\wedge\big[\mathrm{Fr}(K\neq n)=0\big]$ where

n is some nonnegative integer, then we can perform an analysis in the frequency system that shows termination as well; the summary assertion for the loop is

$$[\mathrm{Fr}([K < 0] \vee [K > n]) = 0] \wedge \bigwedge_{0 \le k \le n} [\mathrm{Fr}(K = k) = 1],$$

as we have seen before. But now suppose that we work in the frequency system with the input assertion $\mathrm{Fr}(\text{TRUE}) = 1$, which doesn't explicitly describe the probabilistic distribution of K. There really isn't anything that we can do; it is hopeless to try to construct an informative summary assertion unless we can talk about the distribution of K on input. It would be enough to know K's input distribution in symbolic form, say from an input assertion of the form

$$\bigwedge_{k} [\mathrm{Fr}(K = k) = a_k]$$

for an unspecified sequence $\langle a_k \rangle$, but we have to know something more than $\mathrm{Fr}(\text{TRUE}) = 1$. The frequency system has many sterling characteristics, but power in proving termination is not one of them. In building the frequency system, we aren't focusing on demonstrating the termination of a program about whose computations we know relatively little; rather, we want to devise a formal language to describe all the intricate probabilisitic characteristics of those few programs about whose computations we know almost everything.

The Extremal Assertions.

There is one other issue where a small discussion will perhaps help to clarify the distinction between Floyd-Hoare systems and the frequency system. In a Floyd-Hoare system, if one wants to describe the execution of a process about which one knows nothing, one asserts the predicate TRUE. Since TRUE is always true, this predicate doesn't claim anything. We can find the corresponding null assertion in the frequency system by employing our standard translation, which results in the atomic assertion $\mathrm{Fr}(\neg\,\text{TRUE}) = 0$, that is, $\mathrm{Fr}(\text{FALSE}) = 0$. This atomic assertion clearly holds for every frequentistic state, because every measure ascribes zero mass to the null set. In fact, the atomic assertion $\mathrm{Fr}(\text{FALSE}) = 0$ is equivalent to the assertion TRUE: they both have all of $\mathfrak{F}^{+}$ as their characteristic sets.

The other extremal predicate in a Floyd-Hoare system is the predicate FALSE. If one asserts the truth of this predicate, since FALSE can never be true, one is claiming that control never passes the corresponding point in the flowchart. The characteristic set of FALSE is the empty set, viewed as a subset of $\mathfrak{S}$. Our standard translation shows that the frequentistic assertion corresponding to the predicate FALSE is $\mathrm{Fr}(\text{TRUE}) = 0$. This assertion claims that a total of zero grams of execution mass go by, so that, if control ever does pass that point, it does so only with probability zero. Note, however, that the characteristic set of the assertion $\mathrm{Fr}(\text{TRUE}) = 0$ is not the null set, but is rather the singleton set containing the zero frequentistic state. Therefore, the atomic assertion $\mathrm{Fr}(\text{TRUE}) = 0$ is not equivalent to the assertion FALSE.

On the other hand, there are atomic assertions that are equivalent to the assertion FALSE. One example is the atomic assertion $\mathrm{Fr}(\text{FALSE}) = 1$; since the null subset of $\mathfrak{I}$ has zero mass in any measure, this assertion cannot possibly hold for any frequentistic state. Thus, the characteristic set of the assertiom $\mathrm{Fr}(\text{FALSE}) = 1$, like the characteristic set of the assertion FALSE, is the empty set. These assertions in the frequency system are more false than any predicate, and their existence has certain consequences for the frequency system. In a Floyd-Hoare system, no matter what precondition and postassertion we choose, we can find some program that displays that behavior. After all, the weakest precondition of all is TRUE, and the strongest postassertion is FALSE, but the formula $\{\text{TRUE}\}S\{\text{FALSE}\}$ correctly describes any program S that never terminates. In the frequency system, however, there are precondition and postassertion pairs that could not correctly describe any program. The easiest example is the pair $[\text{TRUE}]S[\text{FALSE}]$, or equivalently

$$\bigl[\mathrm{Fr}(\text{FALSE}) = 0\bigr]\, S\, \bigl[\mathrm{Fr}(\text{FALSE}) = 1\bigr].$$

More generally, any assertion pair whose postassertion demands more total mass than the precondition allows will also be an impossible pair, since no program can generate executions out of thin air.

Programs in the Frequency System.

In the next few sections, we shall get down to brass tacks, and begin to discuss the details of the frequency system. First, we shall sketch out the kind of programs with which it deals. We commented earlier that the S in the formula $[P]S[Q]$ refers to a program in an ALGOL-like programming language; in fact, now that we are allowing our programs to make random choices, we are working exactly with the programs that Kozen called probabilistic **while**-programs. Our program variables, which will be written with upper case italic letters, are assumed either to be of a basic data type, or to represent an array of basic values. The variables can be combined into arithmetic and logical expressions by means of the standard operators and relations, but the evaluation of any expression is assumed to terminate normally. There are statements of various flavors: the empty statement, written

$$\textbf{nothing} \quad ;$$

the deterministic assignment of the expression e to the variable X, written

$$X \leftarrow e \quad ;$$

the random choice of a value for X from the probability distribution F, written

$$X \leftarrow \textbf{Random}_F \quad ;$$

the composition of the statements S and T, written

$$S;\ T \quad ;$$

the conditional statement, written

if P **then** S **else** T **fi** ;

and finally various sorts of single-exit loops, written

while P **do** S **od** ;
loop S **while** P: T **repeat** ;
for J **from** ℓ **to** u **do** S **od** .

And that is all. Extending the frequency system to handle other control structures such as **goto**-statements will be discussed in Chapter 6.

The Assertions of the Frequency System.

The language in which assertions of the frequency system are phrased contains two major layers. The lower layer is basically the predicate calculus, and its job is to describe and analyze the deterministic properties of process state. The upper layer then handles the extension to the probabilistic world.

To be more precise, the lower layer is a first order theory with equality, whose non-logical axioms describe the essential properties of the basic data types. To build such a theory, we start with five different classes of symbols, representing respectively constants, mathematical variables, program variables, functions, and relations. In our text, we shall distinguish the two different flavors of variables by using upper case italic letters for the program variables and lower case italic letters for the mathematical variables. Constants, variables, and function applications are *terms*, and a relation among terms is an *atomic predicate.* The formulas of the lower layer are called *predicates*, and they are built up out of atomic predicates by means of the logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ and the quantifiers $\forall$ and $\exists$. Only a mathematical variable may be bound by a quantifier; program variables must appear freely.

The purpose of a predicate is to specify some of the properties of the deterministic state of a process. The deterministic state of a process, from our current point of view, is merely an assignment of values of the appropriate data types to each of the program variables; as before, let $\mathfrak{D}$ denote the set of all deterministic states. To determine the truth value of a predicate, we need to fix a deterministic state together with an *interpretation* in the normal predicate calculus sense for the constants, mathematical variables, functions, and relations. Of course, we shall only be interested in those interpretations that satisfy the non-logical axioms of our theory; these axioms will determine almost everything about the interpretation, and hence we can afford to ignore the interpretation in what follows. With each predicate, we can then associate a certain subset of $\mathfrak{D}$ called its *characteristic set*, the set of all states in which the predicate holds. We shall restrict ourselves to working with those predicates whose characteristic sets are measurable.

When doing calculations at higher levels in the frequency system, it will often be important to know what the logical relationships are between various predicates, such as when one predicate implies another, or when two predicates are mutually exclusive. To be able to establish these facts formally when necessary, we need a formal system for reasoning about the truth of predicates, that is, for proving theorems. Such systems have been extensively studied, and formal systems of various types have been built to do the job; some common systems are based on natural deduction, or on resolution [25].

On top of this predicate layer, we want to build the language in which probabilistic assertions are to be phrased. This language, and the formal rules for manipulating strings in it, will be called the *assertion calculus*, by analogy with the term "predicate calculus". The main decision to be made when designing the language of the assertion calculus is what level of generality is appropriate. The purpose of an assertion is to specify some of the properties of the frequentistic state of a process, that is, some of the properties of a measure on $\mathfrak{I}$. One way to do this is to specify exactly the value of that measure on particular subsets of $\mathfrak{I}$. This is what an atomic assertion of the form $\mathrm{Fr}(P) = e$ does; it claims that the measure of the characteristic set of the predicate P is given by the expression e. On the other hand, one could also assert more complex relations between the measures of different sets. We have already run across one example of this; recall that in the loop-free world, we discussed the use of probabilistic atomic assertions of the form $\Pr(P) = e$, where this was interpreted to mean

$$\frac{\mathrm{Fr}(P)}{\mathrm{Fr}(\textsc{true})} = e.$$

And much more complex atomic assertions are imaginable; consider, for example, the formula $2\,\mathrm{Fr}(P) < \mathrm{Fr}(Q)^2 + 7$. At this point, we shall sketch out a collection of definitions that take a quite permissive view, and allow even this third example as an atomic assertion. But later, when we begin actually to manipulate assertions formally, we shall restrict ourselves to a special class called the *vanilla* assertions.

Define a *term* of the assertion calculus to be either a constant, or a mathematical variable, or the special real-valued expression $\mathrm{Fr}(P)$ for an arbitrary predicate P, or the result of applying a function to smaller terms. That is, a term in our assertion calculus is just like a term in our predicate calculus, except for two factors: the assertion calculus version allows the new $\mathrm{Fr}(P)$ formulas, and it requires that program variables only appear in the predicates P of such formulas. An *atomic assertion* is then defined to be a relation among terms of the assertion calculus. An atomic assertion may include mathematical variables that appear freely; if we choose a value for each of the free mathematical variables, if any, we can associate with an atomic assertion a *characteristic set*, which will be that subset of $\mathfrak{F}^+$ for which the relation holds. Recall that we are defining the characteristic set of an assertion to be a subset of $\mathfrak{F}^+$ rather than $\mathfrak{F}^*$; that is why the expression $\mathrm{Fr}(P)$ has a value in $\mathbf{R}$ rather than $\mathbf{R}^*$.

An assertion is built up out of atomic assertions with connectives. First, we can use the logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ of the predicate calculus. We can also allow indexed versions of $\wedge$ and $\vee$, which correspond to the predicate calculus quantifiers $\forall$ and $\exists$ respectively. All of these logical connectives are defined in terms of the elementary set theoretic operations on characteristic sets. For example, the characteristic set $\chi(A \wedge B)$ is defined to be the intersection of the characteristic sets $\chi(A)$ and $\chi(B)$.

Furthermore, we can combine atomic assertions with the arithmetic connectives $+$ and $|$. Two frequentistic states c and d in $\mathfrak{F}^+$ can be added together with the addition operator of the vector space $\mathfrak{F}$. The addition operation on assertions is simply set addition in $\mathfrak{F}^+$; that is, the characteristic set $\chi(A+B)$ contains precisely all measures that can be expressed as the sum of one measure in $\chi(A)$ and one in $\chi(B)$. Similarly, the assertion $A \mid P$ denotes the result of a set restriction. Recall that, for any measure c in $\mathfrak{F}^+$, the restricted measure $c \mid P$ ascribes to each measurable subset M of $\mathfrak{D}$ the weight

$$(c \mid P)(M) = c(M \cap \chi(P)).$$

The characteristic set $\chi(A \mid P)$ contains exactly those measures in $\mathfrak{F}^+$ that can be expressed as the restriction to P of a measure in $\chi(A)$. We build *assertions* by combining atomic assertions with these logical and arithmetic connectives. An assertion, like an atomic assertion, is allowed to contain free mathematical variables. If these free variables are given values, the assertion then determines its characteristic set, a subset of $\mathfrak{F}^+$.

Derivations in the Assertion Calculus.

In the last section, we were very liberal in our definitions, and allowed quite a wide class of assertions. But there is more to the assertion calculus than just the language in which assertions are written. In doing formal derivations in the frequency system, we shall want to verify several different classes of facts about assertions. First and most obvious, we shall want to check on occasion whether an assertion has all of $\mathfrak{F}^+$ as its characteristic set, that is, whether it is equivalent to TRUE. Suppose, for example, that we are at a point in a program where the assertion A is known to hold, and we want to know if it is also legitimate to claim the truth of assertion B. This inherited claim will be legitimate if and only if the characteristic set of A is a subset of the characteristic set of B, that is, if and only if the assertion $A \Rightarrow B$ simplifies to the assertion TRUE. In order to check out this kind of fact, we have to develop a set of formal manipulation rules for the assertion calculus, like those for the predicate calculus, in order to distinguish the TRUE assertions. This task might be called *frequency theorem proving.*

Notice, by the way, that some cases of frequency theorem proving are not at all elementary. First, since we have chosen to have an assertion describe a subset of $\mathfrak{F}^+$ rather than of $\mathfrak{F}^*$, note that the assertion

$$\bigwedge_k [\mathrm{Fr}(K = k) = 1]$$

is actually equivalent to the assertion FALSE; that is, its characteristic set is empty. Secondly, consider the assertion

$$\left[\bigwedge_j [\mathrm{Fr}(J = j) = p_j]\right] \wedge \left[\bigwedge_k [\mathrm{Fr}(K = k) = q_k]\right],$$

where the p_j and q_k are some constants. Essentially, this assertion specifies the marginal frequency distributions of both J and K. It will have a non-null characteristic set if and only if there exists some joint distribution for J and K that generates the specified marginals, and this is not an easy thing to determine.

In addition to frequency theorem proving, it will turn out that we shall need to determine whether some of the assertions that arise in our arguments have a certain closure property. When we get around to specifying the rule that handles **while**-loops, we shall have to demand that the output assertion of the loop has this closure property, in order to guarantee the soundness of the While Rule. In particular, we shall have to demand that the characteristic set of the output assertion be *closed* in the usual sense. Any Cauchy sequence in $\mathfrak{F}$ converges to a limit, because $\mathfrak{F}$ is a Banach space. A subset of $\mathfrak{F}$ is *closed* if and only if it contains the limit points of every Cauchy sequence that it contains. We shall call an assertion *closed* exactly when its characteristic set is closed.

Working in the frequency system will present us with a dual challenge in the domain of reasoning about assertions; we must be able to do some frequency theorem proving, and we also must be able to check that some assertions are closed. Furthermore, we have to be able to do this reasoning by formal symbol manipulation, of course. Building symbol manipulation rules that could tackle these jobs in general seems like a formidable job, and we shall not undertake it here. Instead, we shall take advantage of the fact that all of the assertions that will arise in our examples have a certain very special form, and this will allow us to get by with a relatively simple assertion manipulation capability. In particular, we shall restrict ourselves to describing the behavior of programs with *vanilla assertions.* A vanilla atomic assertion, or *clause*, is an atomic assertion of the special form $\mathrm{Fr}(P) = e$ that we worked with earlier; here P is an arbitrary predicate, and e is a real-valued expression, and the clause has as its characteristic set exactly those measures in $\mathfrak{F}^+$ that ascribe mass e to the set $\chi(P)$. An assertion is *vanilla* if it consists of a conjunction of clauses. This conjunction can involve the use of explicit $\wedge$'s, unbounded universal quantification, or even bounded universal quantification, which means formulas of the form $\bigwedge_{i:R} A$ where A is a vanilla assertion in which i appears freely, and R is a predicate in which i appears freely but no program variable appears. A vanilla assertion is a recipe for a finite or infinite collection of clauses, and the recipe is concrete enough that only a theorem prover for the predicate calculus is needed to decipher it.

Note that the conditions that make an assertion vanilla only discuss the structure of that assertion at the assertion calculus level. There is no restriction on the type of predicates P that can appear in clauses $\mathrm{Fr}(P) = e$, for example; arbitrary predicates are permissible. This has the pleasant consequence that our restriction to vanilla assertions doesn't affect the embedding of Floyd-Hoare analyses into the frequency system. Recall that the analog of the Floyd-Hoard predicate P is the atomic assertion $\mathrm{Fr}(\neg P) = 0$. This is a single clause, and thus also a vanilla assertion, no matter what the structure of P might be.

Working with Vanilla Assertions.

If we limit ourselves to making only vanilla assertions about the behavior of a program, then only very special cases of frequency theorem proving and of checking for closure will ever arise. In fact, we can eliminate the latter problem at once, by noting that every vanilla assertion is closed. Consider first a clause $\mathrm{Fr}(P) = e$. Let $\chi(P)$ denote P's characteristic subset of $\mathfrak{I}$, let $\langle c_n \rangle$ denote a Cauchy sequence in $\mathfrak{F}$, let c_∞ denote its limit, and suppose that c_n satisfies the clause $\mathrm{Fr}(P) = e$ for all n. Then, for all n, we must have $c_n\bigl(\chi(P)\bigr) = e$, which implies that $c_\infty\bigl(\chi(p)\bigr) = e$ as well. Therefore the limiting measure c_∞ will also satisfy the clause. This same reasoning applies to most forms of atomic assertions that describe equalities or weak inequalities between expressions involving $\mathrm{Fr}(P)$ terms. But strict inequalitites in atomic assertions, such as $\mathrm{Fr}(P) < e$, in general destroy closure.

Since a vanilla assertion is simply a conjunction of clauses, its characteristic set is simply the intersection of the characteristic sets of the clauses. Since arbitrary (even uncountable) intersections of closed sets are closed, we can deduce that every vanilla assertion is closed. The ease of this proof is one of the great benefits of restricting ourselves to vanilla assertions. Note that infinite unions, or even a single complementation, can destroy closure.

The restriction to vanilla assertions also eases the job of frequency theorem proving. Basically four kinds of challenges will arise in the course of our program analyses in the frequency system. First, as we mentioned before, we shall sometimes want to know when one assertion implies another. With our restriction to vanilla assertions, this means that we are interested in the truth of some formulas of the form $A \Rightarrow B$ for vanilla assertions A and B. In general, this problem even with the vanilla restriction is quite subtle. For example, one way of demonstrating the implication is to show that A is actually equivalent to FALSE; we discussed several cases of assertions that were false for quite subtle reasons, and those hard examples were in fact vanilla assertions. However, in many cases, including those that will arise in our example programs, the implication $A \Rightarrow B$ can be demonstrated by fairly elementary reasoning. Certainly, the operation of removing some of the clauses from A will only make it weaker, so the implication holds if every clause in B also appears in A. In addition, the additivity of measures allows us to draw some conclusions. For example, if A contains the clauses $\mathrm{Fr}(P) = e$ and $\mathrm{Fr}(Q) = f$, and if the predicates P and Q can be shown to be mutually exclusive by a derivation in the predicate calculus, then B is justified in containing the clause $\mathrm{Fr}(P \vee Q) = e + f$. Clauses with a zero right-hand side allow another form of deduction. If A contains the clause $\mathrm{Fr}(P) = 0$, then B can contain not only this clause, but also the clause $\mathrm{Fr}(Q) = 0$ for any predicate Q such that $Q \Rightarrow P$ is a theorem of the predicate calculus; subsets of measure zero sets must have measure zero. We shall assume without more detailed specification a collection of formal rules for verifying implications between vanilla assertions that has the ability to perform these elementary types of reasoning.

The next two kinds of theorems that we shall meet arise at the forks and joins of the plumbing network. At forks, we are presented with formulas of the form $A \mid P \Rightarrow B$, while at joins, we find $A + B \Rightarrow C$; here, A, B, and C denote vanilla assertions, and P denotes an arbitrary predicate. Once again, there are subtle theorems of these forms, but at least some such theorems succumb to elementary reasoning. Let us consider the join case first; what clauses can we justify having in C if we want the formula $A + B \Rightarrow C$ to hold? Or, in programming terms, what clauses describe the out-branch of a join of which A and B describe the in-branches? The basic fact at work is the following: if A contains the cluase $\mathrm{Fr}(P) = e$ and B contains the clause $\mathrm{Fr}(P) = f$, then C may contain the clause $\mathrm{Fr}(P) = e + f$. This derivation just asserts the conservation of mass for P-colored pellets at the join. Sometimes, other sorts of reasoning must be performed before this basic insight can be applied. For example, suppose that A contains the clause $\mathrm{Fr}(P) = e$ and B the clause $\mathrm{Fr}(Q) = f$; if the predicates P and Q can be proved equivalent by reasoning in the predicate calculus, we can add the clause $\mathrm{Fr}(P) = f$ to B, and then apply the previous insight to deduce that $\mathrm{Fr}(P) = e+f$ belongs in C. Or it might happen that A describes the mass of an event while B describes the mass of each of the subsets of a partition of that event. Then, we can replace B by an assertion B' that describes the mass of the whole event at once, and such that $B \Rightarrow B'$; after this replacement, our basic addition insight will apply.

It can happen that a clause in one of the summands does not affect the result at all. For example, if A contains the clause $\mathrm{Fr}(P) = e$ but B does not specify the measure of the characteristic set of P, there is nothing we can do in the realm of vanilla assertions. We could assert that $\mathrm{Fr}(P) \geq e$ in C, since B will only increase the amount of P-colored mass, but this leaves the realm of vanilla assertions.

Similar issues come up when we consider a fork in the network. If A describes the in-branch of a fork that tests the predicate Q, then $A \mid Q$ and $A \mid \neg Q$ describe the TRUE and FALSE out-branches respectively. Consider a clause $\mathrm{Fr}(P) = e$ in A. If P implies Q in the predicate calculus, then all of the P-colored mass that A describes will follow the TRUE out-branch; thus, we can add the clause $\mathrm{Fr}(P) = e$ to $A \mid Q$, and the clause $\mathrm{Fr}(P) = 0$ to $A \mid \neg Q$. On the other hand, if P implies $\neg Q$, we can do the reverse, adding $\mathrm{Fr}(P) = e$ to $A \mid \neg Q$ and $\mathrm{Fr}(P) = 0$ to $A \mid Q$. In fact, independent of what clauses are in P, we can add the clause $\mathrm{Fr}(\neg Q) = 0$ to $A \mid Q$, and the clause $\mathrm{Fr}(Q) = 0$ to $A \mid \neg Q$; these just assert the accuracy of the test on Q. Again, this basic splitting insight can be decorated with extra deduction steps that involve the elementary properties of measures and of the real numbers.

As long as we restrict ourselves to vanilla assertions, we never need to worry about closure, and the three types of frequency theorems that we have discussed so far can be handled, at least in many cases, by elementary formal reasoning. We shall have a little bit more trouble handling the fourth kind of frequency theorem that will arise during our use of the frequency system. The fourth type of theorem involves showing that an assertion is not equivalent to FALSE.

Checking Feasibility.

The fourth kind of frequency theorem that will come up in our use of the frequency system is another facet of the problems engendered by loops. In order to avoid time bombs, we shall have to guarantee that the summary assertions that describe loops have the property that their characteristic sets are not empty. Call an assertion *feasible* when there is a finite measure that satisfies it. With this terminology, we shall demand that all candidates for summary assertions be feasible. The problem of checking feasibility corresponds to the assertion calculus problem of determining whether or not the assertion A is equivalent to FALSE. As some of our previous examples demonstrated, this can be quite a subtle problem.

Once again we shall only attempt to solve certain special cases. Suppose that A is a vanilla assertion; what properties of A will guarantee the existence of a finite measure that satisfies all of the clauses of A? If the clauses of A describe mutually exclusive events, and if the sum of all of the right-hand sides of the clauses of A is finite, then the assertion A is not equivalent to FALSE. For example, consider the assertion

$$A = \bigwedge_{j,k}\bigl[\mathrm{Fr}([J = j] \wedge [K = k]) = p_{j,k}\bigr]$$

where the coefficients $p_{j,k}$ are nonnegative and have a finite sum. We can build an explicit finite frequentistic state that satisfies the assertion A by the following process: For each j and k, choose a deterministic state in which $J = j$ and $K = k$, and in which the other components of the process state, if any, are chosen arbitrarily. Ascribe to each of these deterministic states the corresponding mass $p_{j,k}$, and ascribe zero mass to every other state. The result is a finite measure that satisfies the assertion A.

When we are working with discrete problems, the reasoning behind this example will often be enough. We shall call an assertion A *disjoint vanilla* if it is vanilla, and also satisfies the following three conditions: (i) any two distinct clauses in A describe mutually exclusive events, (ii) the sum of all of the right-hand sides of the clauses in A is finite, and (iii) no clause in A ascribes nonzero mass to a predicate whose characteristic subset of $\mathfrak{I}$ is the empty set. (The third condition is a technicality that was omitted in the previous paragraph.) These three conditions can often be checked by formal reasoning, and the argument above shows that every disjoint vanilla assertion is feasible.

When we deal with variables whose distributions are nondiscrete, however, we cannot afford to limit ourselves to disjoint vanilla assertions. For example, suppose that we want to describe a frequentistic state with a total of 1 gram of mass, in which the real-valued program variable X is uniformly distributed on the interval $[0, 1)$. Since X is a continuous variable, the correct right-hand side for any frequentistic assertion of the form $\mathrm{Fr}(X = x) = e$ would have to be 0. Instead of describing the exact value of X, we must instead give the frequencies with which X lies in certain subsets of the real line. For example, one assertion that does the job would be

$$\bigl[\mathrm{Fr}([X < 0] \vee [X \geq 1]) = 0\bigr] \wedge \bigwedge_{0 \leq x \leq 1} \bigl[\mathrm{Fr}(0 \leq X < x) = x\bigr].$$

We could, at our option, extend the conjunction to include one atomic assertion for every measurable subset M of the reals, which stated that

$$\mathrm{Fr}(X \in M) = \mu(M \cap [0,1)),$$

where μ here denotes Lesbegue measure. The difficulty is that, in any of these schemes, the atomic assertions must describe the masses of sets that are not disjoint; hence, the resulting assertions, while vanilla, are not disjoint vanilla.

The problem of checking the feasibility of an assertion about nondiscrete variables is hard to solve in any general way. We shall take advantage of the fact that the specific cases that arise in our examples have a special form. In particular, we shall only deal with real-valued nondiscrete variables, and we shall describe their distributions in terms of the associated cumulative distribution functions or densities. Every cumulative distribution function determines a corresponding measure on the Borel sets [10], and we shall appeal to the existence of these measures to establish the feasibility of assertions. We postpone the details for now.

The Rules of the Frequency System.

We have settled on the structure of predicates and of assertions, and discussed the issues connected with formally deriving the results that we shall need in the predicate and assertion calculi. We have thus finally arrived at the stage where we can give the rules of the frequency system. It is the purpose of these rules and axiom schemata to distinguish a certain collection of formulas $[A]S[B]$ of the frequency system as *theorems*, written $\vdash [A]S[B]$. Furthermore, we want to show that every theorem will be a semantically true statement. We shall continue to denote the characteristic set of an assertion A by $\chi(A)$. With this notation, a formula $[A]S[B]$ is semantically true if and only if the relation $f(\chi(A)) \subseteq \chi(B)$ holds, where f is the continuous linear map from $\mathcal{F}$ to $\mathcal{F}$ that formally defines the meaning of the program S in Kozen's semantics [22]. Each syntactic construct of our programming language will need to be handled somehow in the frequency system, and we shall consider them in turn. Most of the hard work will concern the While Rule, as one might expect. (Some aspects of the following presentation are taken from a dicussion of the Floyd-Hoare rules by Ole-Johan Dahl [5].)

The Rules of Consequence.

First, we have the rules of consequence,

$$\frac{\vdash A \Rightarrow B, \quad \vdash [B]S[C]}{\vdash [A]S[C]} \quad \text{and} \quad \frac{\vdash [A]S[B], \quad \vdash B \Rightarrow C}{\vdash [A]S[C]},$$

which allow us to weaken our postassertions and strengthen our preconditions if we so desire. These rules are the same as in Floyd-Hoare systems. As we mentioned in Chapter 1, the formulas above the line indicate the premises of the deduction step, and the formula below the line indicates the corresponding conclusion. Notice that these rules have premises of two different types. In the first rule, for example, the premise $\vdash A \Rightarrow B$ indicates that the assertion $A \Rightarrow B$

should be a theorem of the assertion calculus in order for this rule to apply, while the premise $\vdash [B]S[C]$ indicates that the augmented program $[B]S[C]$ should be a theorem of the frequency system.

The soundness of the Rules of Consequence follows immediately from the definition of semantic truth. Again, let us take the first rule as an example. The truth of the first premise implies the relation $\chi(A) \subseteq \chi(B)$; the truth of the second premise implies the relation $f(\chi(B)) \subseteq \chi(C)$, where f denotes the semantic interpretation of the program S. Putting these together, we deduce that $f(\chi(A)) \subseteq \chi(C)$, which demonstrates the semantic truth of the conclusion.

The Axiom Schema of the Empty Statement.

The empty statement has, instead of a rule, a family of axioms associated with it. In particular, we are allowed to conclude without any premises

$$\vdash [A]\ \mathbf{nothing}\ [A],$$

where A represents any assertion. This is also no change from the Floyd-Hoare situation.

Kozen's semantics interprets the empty statement as the identity function from $\mathfrak{F}$ to $\mathfrak{F}$; therefore, the Axiom Schema of the Empty Statement is trivially sound.

The Assignment Axiom Schema.

Like the empty statement, the assignment statement has an associated axiom schema instead of a rule. Recall that in a Floyd-Hoare system, there are axioms that define the behavior of an assignment statement either by going forward or by going backward. The backward, that is, from right to left, axioms are the simplest, stating that

$$\vdash \{P^X_e\}\ X \leftarrow e\ \{P\}.$$

Here, the assignment statement is setting the program variable X to the current value of the expression e, while P represents an arbitrary predicate and P^X_e represents the results of textually substituting e for X everywhere that X appears in P. When thinking about axioms for the assignment statement, it is best to work with an example where the expression e does depend on X, but not in a one-to-one manner; we shall take the assignment $X \leftarrow X^2 + 1$ as our example. One possible instance of the above axiom schema in this case is the Floyd-Hoare theorem

$$\vdash \{(X^2+1) = 10\}\ X \leftarrow X^2 + 1\ \{X = 10\};$$

using our knowledge about arithmetic on the integers, we could rephrase the precondition equivalently as $[X = 3] \vee [X = -3]$, to arrive at the theorem

$$\vdash \{[X = 3] \vee [X = -3]\}\ X \leftarrow X^2 + 1\ \{X = 10\}$$

There is also a somewhat more complex axiom schema that works forward, from left to right:

$$\vdash \{P\}\ X \leftarrow e\ \{(\exists y)([X = e^X_y] \wedge P^X_y)\}.$$

In this schema, the value of y that exists is the value that X had before the assignment. For example, we could deduce

$$\vdash \{X = 3\}\ X \leftarrow X^2 + 1\ \{(\exists\, y)([X = y^2 + 1] \wedge [y = 3])\}.$$

Simplifying a little, this is just the Floyd-Hoare theorem

$$\vdash \{X = 3\}\ X \leftarrow X^2 + 1\ \{X = 10\};$$

note that this formula, while a theorem, is less informative than the result of the backward analysis above, since its precondition is stronger.

The obvious way to produce corresponding axiom schemata for the frequency system is to use the above techniques on the predicates that are embedded within assertions. Recall that an assertion is built up out of real-valued terms of the form $\mathrm{Fr}(P)$, where P denotes a Floyd-Hoare predicate; furthermore, program variables in assertions may only appear inside these predicates. Since only the program variables are affected by an assignment, we might expect that performing the above manipulations on the embedded predicates would do the job. For the backward rule, this is indeed the case, and it gives us a corresponding Backward Assignment Axiom Schema for the frequency system:

$$\vdash [A_e^X]\ X \leftarrow e\ [A].$$

We can merely state that all of the X's in the assertion A should be replaced by e's, since all of the X's will lie inside predicates. An instance of this axiom schema is the frequency system theorem

$$\vdash [\mathrm{Fr}([(X^2 + 1) = 10]) = 7]\ X \leftarrow X^2 + 1\ [\mathrm{Fr}(X = 10) = 7].$$

Rephrasing the precondition, we can write this theorem

$$\vdash [\mathrm{Fr}([X = 3] \vee [X = -3]) = 7]\ X \leftarrow X^2 + 1\ [\mathrm{Fr}(X = 10) = 7].$$

Before we show that the Backward Assignment Axiom Schema is sound, let us consider the Forward Schema that would result from performing the forward Floyd-Hoare predicate transformation to all embedded predicates. Working with the same example again, a Forward Schema would suggest that the formula

$$[\mathrm{Fr}(X = 3) = 7]\ X \leftarrow X^2 + 1\ [\mathrm{Fr}((\exists\, y)([X = y^2 + 1] \wedge [y = 3])) = 7]$$

should be a theorem of the frequency system, and hence also the equivalent formula

$$[\mathrm{Fr}(X = 3) = 7]\ X \leftarrow X^2 + 1\ [\mathrm{Fr}(X = 10) = 7].$$

But this formula is not necessarily true! The input assertion does not rule out the possibility of the existence of positive mass in which $X = -3$, and such mass would cause there to be a total of more than seven grams of output mass in which $X = 10$. Thus, although the technique of moving from left to right over an assignment by existentially quantifying over the old value of the variable generates a sound rule in the Floyd-Hoare world, it is not accurate enough for the frequency system. The intuitive failing of the forward technique is that the preconditions in its theorems are not the weakest possible. Since the forward schema does not pan out, we shall drop the adjective "Backward" from the name of the Backward Assignment Axiom Schema.

The Assignment Axiom Schema is sound, and the proof hinges on the fact that the preconditions generated by the backward Floyd-Hoare technique are the weakest possible. To see this in more detail, we must first recall the interpretation given the assignment statement $X \leftarrow e$ by Kozen's semantics. Let $v: \mathfrak{D} \rightarrow \mathfrak{D}$ be the function that describes the effect of the assignment $X \leftarrow e$ on a deterministic input state; that is, v replaces the current X-coordinate of the state vector by the appropriate new value e. According to Kozen's semantics, if the frequentistic state of a process before the assignment $X \leftarrow e$ is the measure c, then the frequentistic state after the assignment will be the measure $c \circ v^{-1}$, where v^{-1} is defined as usual by

$$v^{-1}(M) = \{m \mid v(m) \in M\}.$$

The critical fact for us to observe is the following:

$$v^{-1}(\chi(P)) = \chi(P_e^X). \tag{4.1}$$

We can restate this insight in words by observing that, for any deterministic state m, the predicate P holds for $v(m)$ if and only if the predicate P_e^X holds for m itself, since the effect of v is precisely to replace the current value of X by e. In fancier terminology, we are just observing that the preconditions generated by the backward Floyd-Hoare technique are the weakest possible.

Equation (4.1) is a deterministic fact; but it happens to be just what we need to show the soundness of our frequentistic Assignment Axiom Schema. Consider one of the axioms of this schema, say

$$\vdash [A_e^X]\; X \leftarrow e\; [A].$$

Inside the postassertion A are various real-valued terms of the form $\mathrm{Fr}(P)$ for various predicates P. The corresponding term in the precondition A_e^X will be precisely $\mathrm{Fr}(P_e^X)$. Now, consider any input frequentistic state c for the assignment statement that satisfies the precondition A_e^X (if the precondition is not feasible, we are done trivially). Assuming that c satisfies A_e^X merely means that, if we substitute for the terms $\mathrm{Fr}(P_e^X)$ the corresponding real numbers

$$c(\chi(P_e^X)), \tag{4.2}$$

the resulting formula simplifies to TRUE.

According to Kozen's semantics, the output frequentistic state corresponding to the input state c will be $c \circ v^{-1}$. Our task is to show that this state satisifes the output assertion A. Of course, the measure $c \circ v^{-1}$ satisfies A if and only if the formula obtained by substituting the real numbers

$$c \circ v^{-1}(\chi(P)) = c(v^{-1}(\chi(P))) \tag{4.3}$$

for the terms $\mathrm{Fr}(P)$ simplifies to TRUE. But Equation (4.1) shows that the quantities (4.2) and (4.3) must be equal. Therefore, $c \circ v^{-1}$ will satisfy A whenever c satisfies A_e^X, and we have shown that the axioms produced by the Assignment Axiom Schema are true.

There are some caveats associated with the use of the Assignment Axiom Schema. First, to make the argument above correct, we must invoke our assumption that the evaluation of every expression e in our programming language is guaranteed to terminate normally. In addition, we must make several qualifications about the program's naming structures. In order for the schema to be sound, syntactically distinct variables must refer to disjoint storage locations, and hence be also semantically distinct; in more technical language, there must be no *aliasing*. Also, assignments to an array element must be treated as assignments to the entire array. For example, the assignment $X[I] \leftarrow e$ would be handled as if it were actually the array assignment $X \leftarrow (X \mid I \mid e)$, where this latter expression, a *change triple*, denotes X with its Ith component changed to be e. We shall not pay too much attention to these sorts of problems, since the issues involved and the possible solutions are the same in the frequency system as they are in the deterministic world.

Most of the derivations in the frequency system are best thought of as taking place from left to right, following the direction of the program flow. For example, whenever we apply the addition and restriction operations, we are modeling the forward flow of control. The Assignment Axiom Schema thus goes against the grain somewhat by moving from right to left. Suppose, for example, that the state of the world before the assignment $J \leftarrow J+1$ is described by the assertion

$$\bigwedge_j [\mathrm{Fr}(J = j) = p_j]$$

for some parameters p_j, and suppose that we would like to describe the state of things after the assignment. One good way to proceed is first to massage the precondition a little by replacing the mathematical variable j throughout by $j - 1$. This generates the equivalent assertion

$$\bigwedge_j [\mathrm{Fr}(J = j - 1) = p_{j-1}],$$

which we could also write

$$\bigwedge_j [\mathrm{Fr}(J + 1 = j) = p_{j-1}].$$

Now we can see how to use the Assignment Axiom Schema; in particular, one instance of that schema is the axiom

$$\vdash \left[\bigwedge_j [\mathrm{Fr}(J + 1 = j) = p_{j-1}]\right] J \leftarrow J + 1 \left[\bigwedge_j [\mathrm{Fr}(J = j) = p_{j-1}]\right],$$

where J is replaced by $J+1$ in the assertions, moving from right to left. Therefore, we may conclude that the assertion

$$\bigwedge_j [\mathrm{Fr}(J=j)=p_{j-1}]$$

holds after the assignment. As we do more examples, we shall get more adept at this sort of manipulation.

It is often necessary to use the Rules of Consequence and some frequency theorem proving to get a useful result out of the Assignment Axiom Schema. Suppose, for example, that we try to carry the precondition $\mathrm{Fr}(\text{TRUE})=1$ through the assignment $K \leftarrow 0$. The best thing to do is first to replace the precondition with the equivalent assertion

$$[\mathrm{Fr}(0=0)=1] \wedge [\mathrm{Fr}(0 \neq 0)=0].$$

Then, an appropriate instance of the Assignment Schema will allow us to conclude that the assertion

$$[\mathrm{Fr}(K=0)=1] \wedge [\mathrm{Fr}(K \neq 0)=0]$$

holds after the assignment, as we would expect.

The Axiom Schema of Random Choice.

Unfortunately, the operators of our current assertion calculus are not powerful enough to express in a clean way the legal derivations concerning random choices; thus, the title of this section is really a lie. Rather than give an axiom schema for the random assignment statement, we shall just discuss a certain collection of elementary axioms about random choices.

Consider the random assignment $X \leftarrow \mathrm{Random}_F$, and let D denote the set of all possible values of the data type of X. The subscript F denotes a probability distribution on D, that is, a function that ascribes mass $F(M)$ to each measurable subset M of D. By calling F a probability distribution rather than a frequency distribution, we are assuming that the norm of F is 1; that is, F is a positive measure on D with $F(D)=1$. Furthermore, suppose that we are about to execute the random assignment, and that our current state is described by the assertion

$$\bigwedge_j [\mathrm{Fr}(J=j)=p_j].$$

What postassertion can we make, to describe the state of the program after the random choice? One might initially consider the assertion

$$\Bigl[\bigwedge_j [\mathrm{Fr}(J=j)=p_j]\Bigr] \wedge \Bigl[\bigwedge_M [\mathrm{Fr}(X \in M)=F(M)]\Bigr],$$

but this candidate has two problems. The first is that the total amount of mass entering the choice, which is just $\sum_j p_j$, might not be 1. If it isn't, then we would have to rescale the distribution

F appropriately. But a far worse problem also pertains: this suggested postassertion does not give any information about the joint distribution of J and X. It is part of the specification of the random assignment statement that the random choice is made independently of everything that has happened so far in the execution of the program, and it is important that our axioms for random assignment reflect that. We can cure both of these problems at once by choosing the postassertion

$$\bigwedge_{j,M}\left[\mathrm{Fr}(J=j, X\in M)=p_jF(M)\right].$$

This postassertion shows that every pellet that comes through the random assignment is broken up into pieces according to the distribution F, which is what we intended.

By the way, this assertion also introduces a new notation; we would have written this assertion previously as

$$\bigwedge_{j,M}\left[\mathrm{Fr}([J=j]\wedge[X\in M])=p_jF(M)\right].$$

As our formulas get more and more complex, we shall sometimes use a comma instead of the symbol $\wedge$ to indicate an "and". The advantage is not that the comma is thinner, but that the use of the comma allows us to get by with fewer parentheses.

We shall be satisfied with the axioms for random assignment statements that follow the pattern above. We shall arrange that the precondition of the random assignment is a vanilla assertion that does not mention the variable whose value is being randomly chosen. Then, for each atomic assertion $\mathrm{Fr}(P)=e$ in the precondition, we shall allow ourselves to add to the postassertion the family of atomic assertions

$$\bigwedge_{M}\left[\mathrm{Fr}(P, X\in M)=eF(M)\right].$$

The conjunction here is over all measurable subsets M of D. In the common case where the distribution F is discrete, we can get by with a simpler form of postassertion; rather than handling every measurable set, it is enough to handle each mass point. For example, suppose that F ascribes mass f_k to each integer k. Then, the clause $\mathrm{Fr}(P)=e$ in the precondition will generate the clauses

$$\bigwedge_{k}\left[\mathrm{Fr}(P, X=k)=ef_k\right]$$

in the postassertion.

The process given above will allow us to construct an axiom for a random assignment statement as long as the desired precondition is both a vanilla assertion, and does not mention the variable whose value is being chosen. It remains to show that the axioms so produced are in fact true. To see this, we need to refer once again to Kozen's semantics. For notational simplicity, let us assume that the variable whose value is being randomly chosen is actually the first program variable X_1 of data type D_1. We are considering the choice statement $X_1 \leftarrow \mathbf{Random}_F$, where F denotes a probability distribution on D_1. Recall that Kozen interprets this statement as the linear map f from $\mathcal{F}$ to $\mathcal{F}$ that satisfies the identity

$$\bigl(f(c)\bigr)(M_1 \times \cdots \times M_n) = c(D_1 \times M_2 \times \cdots \times M_n)F(M_1) \tag{4.4}$$

for all measures c in $\mathcal{F}$ and measurable subsets M_j of D_j for $1 \le j \le n$.

Consider a clause $\mathrm{Fr}(P) = e$ in a vanilla precondition for the random assignment. If P does not mention the variable X_1, then the characteristic set $\chi(P)$ can be expressed as a direct product

$$\chi(P) = D_1 \times Z \qquad \text{for} \qquad Z \subseteq D_2 \times \cdots \times D_n.$$

If c is any frequentistic input state that satisfies the precondition, we must have

$$c(\chi(P)) = c(D_1 \times Z) = e. \tag{4.5}$$

The precondition clause $\mathrm{Fr}(P) = e$ will cause our axiom building process to put into the postassertion the collection of clauses

$$\bigwedge_{M_1}[\mathrm{Fr}(P, X_1 \in M_1) = eF(M_1)]. \tag{4.6}$$

The characteristic sets of the predicates in these clauses are quite easy to compute; we have

$$\chi(P \wedge [X_1 \in M_1]) = \chi(P) \cap \chi(X_1 \in M_1) = M_1 \times Z.$$

In order for the clauses (4.6) to hold for the output state $f(c)$, we must show that

$$\bigl(f(c)\bigr)(M_1 \times Z) = eF(M_1).$$

But if Equation (4.4) holds for all M_2, M_3, …, M_n, it must also be the case that

$$\bigl(f(c)\bigr)(M_1 \times Z) = c(D_1 \times Z)F(M_1).$$

In light of Equation (4.5), we are done. Therefore, our axiom building process for random choices is sound.

The Composition Rule.

The Rule of Composition is the same in the frequency system as in the Floyd-Hoare world:

$$\frac{\vdash [A]\, S\, [B], \quad \vdash [B]\, T\, [C]}{\vdash [A]\, S;\ T\, [C]}.$$

The proof of soundness is easy. If f and g represent the interpretations of the programs S and T respectively, then the first premise shows that $f(\chi(A)) \subseteq \chi(B)$, while the second premise shows that $g(\chi(B)) \subseteq \chi(C)$. These two inclusions imply the result $g \circ f(\chi(A)) \subseteq \chi(C)$, which finishes the job, because Kozen's semantics interprets the statement "S; T" as the composed function $g \circ f$.

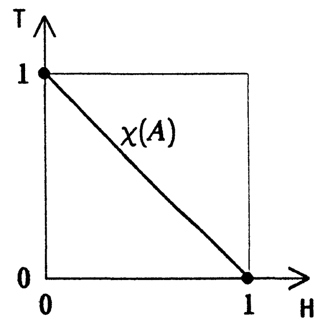

Figure 4.1. The space $\mathcal{F}$ of all measures on $\mathcal{D} = \{\mathsf{H}, \mathsf{T}\}$.

The Conditional Rule.

The Conditional Rule in the frequency system differs from the corresponding rule in Floyd-Hoare, but that difference is simply a reflection of the different operators that the two systems use to encode the actions of forks and joins. Indeed, we gave the Conditional Rule for the frequency system way back in Chapter 2; to reiterate, the rule is

$$\frac{\vdash [A \mid P]\, S\, [B], \quad \vdash [A \mid \neg P]\, T\, [C]}{\vdash [A]\ \textbf{if}\ P\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}\ [B + C]}.$$

This rule is easily seen to be sound. If the conditional statement in the conclusion is executed beginning in a frequentistic state a, Kozen's semantics defines the resulting output state to be $f(a \mid P) + g(a \mid \neg P)$, where f and g are the interpretations of S and T respectively. The premises of the Conditional Rule show that, for any measure a in $\chi(A)$, the memberships $f(a \mid P) \in \chi(B)$ and $g(a \mid \neg P) \in \chi(C)$ must hold; adding these together finishes the proof.

Throughout our arguments, we have been concerned with making sure that the formal system that we are building is sound: a sound frequency system, built on top of sound assertion and predicate calculi. Another desirable property for a formal system is completeness; a formal system is *complete* if all semantically true formulas are actually theorems. It is interesting to note that our Conditional Rule is not complete.

Consider the program

UselessTest: **if** $X = \mathsf{H}$ **then nothing else nothing fi.**

The variable X in this program, like the X in the program CoinFlip, stores the state of a coin, and hence is restricted to the two values H and T standing for heads and tails respectively. Assuming that X is the only component of the process state, a process executing the UselessTest program has exactly two possible deterministic states, the states $X = \mathsf{H}$ and $X = \mathsf{T}$. Therefore, a frequentistic state here is just a pair of nonnegative real numbers, each of which gives the frequency of one of these two deterministic states. The vector space $\mathcal{F}$ is two-dimensional, and its positive cone $\mathcal{F}^+$ is just the first quadrant.

Suppose that we choose as a precondition for the UselessTest program the assertion

$$A = \big[\mathrm{Fr}(\textsc{true}) = 1\big]. \tag{4.7}$$

Figure 4.1 shows the characteristic set $\chi(A)$ of the assertion A; the abscissa gives the frequency of $X = \mathsf{H}$, while the ordinate gives the frequency of $X = \mathsf{T}$. If we execute UselessTest with any frequentistic state a in $\chi(A)$ as input, we will get the same state a back out as output. In detail, the test of $X = \mathsf{H}$ will split the state a into $a \mid (X = \mathsf{H})$ and $a \mid (X = \mathsf{T})$. This split merely resolves the state a into its H and T components. Both arms of the conditional are empty, so the two components are recombined at the final join to result in the output state a. This verifies that UselessTest is a frequentistic no-op, as we would expect.

We would also expect, therefore, to be able to verify the augmented program

$$[A] \textbf{ if } X = \mathsf{H} \textbf{ then nothing else nothing fi } [A]. \tag{4.8}$$

Unfortunately, our Conditional Rule cannot demonstrate (4.8). We begin with the assertion $A = [\mathrm{Fr}(\mathrm{TRUE}) = 1]$. When we restrict A to the truth of the condition $X = \mathsf{H}$, we get the assertion

$$[A \mid (X = \mathsf{H})] \;=\; [\mathrm{Fr}(X = \mathsf{H}) \leq 1] \wedge [\mathrm{Fr}(X = \mathsf{T}) = 0]. \tag{4.9}$$

This is a not a vanilla assertion, but for our current purposes, that is no problem. The characteristic set of (4.9) is the H-axis from 0 to 1. Similarly, when we restrict A to the falsity of the control test, we get the assertion

$$[A \mid (X = \mathsf{T})] \;=\; [\mathrm{Fr}(X = \mathsf{T}) \leq 1] \wedge [\mathrm{Fr}(X = \mathsf{H}) = 0], \tag{4.10}$$

which has the T-axis from 0 to 1 as its characteristic set. Since both arms of the conditional are empty, the assertions (4.9) and (4.10) also play the roles of B and C in the Conditional Rule. The output assertion that we can give for the UselessTest program is then

$$B + C = [\mathrm{Fr}(X = \mathsf{H}) \leq 1] \wedge [\mathrm{Fr}(X = \mathsf{T}) \leq 1],$$

which has the entire unit square as its characteristic set. The best that our Conditional Rule can say about the program UselessTest is the theorem

$$\vdash [A] \textbf{ if } X = \mathsf{H} \textbf{ then nothing else nothing fi } [B + C].$$

This theorem, while true, is much weaker than the true but unproven formula (4.8).

This example demonstrates that our Conditional Rule is incomplete, and any system based upon it will be incomplete as well. Furthermore, this incompleteness is not a result of some weakness in the underlying assertion calculus or predicate calculus. Instead, it is a result of the clumsiness of set operations. When we project the set $\chi(A)$ down onto a coordinate axis at the beginning of one branch of the conditional, we lose track of the correlation between the H and T components of the points in $\chi(A)$. The set addition at the final join punishes us for this lost information by filling the entire unit square. Thus, the incompleteness of our Conditional Rule has its roots in one of the basic choices behind the frequency system: that assertions should specify sets of frequentistic states.

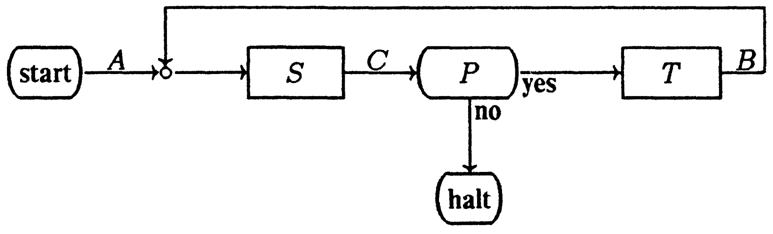

Figure 4.2. A general **repeat**-loop.

What shall we do about this incompleteness? One possible action is to patch it by adjusting the system somewhat. We can handle this particular flavor of incompleteness if we add to the frequency system a second rule for the **if**-statement, which might be called the Irrelevant Conditional Rule:

$$\frac{\vdash [A]\ S\ [B],\quad \vdash [A]\ T\ [B]}{\vdash [A]\ \textbf{if}\ P\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}\ [B]}.$$

The intuition behind this rule is that, if it doesn't matter which branch of the conditional statement is executed, then there is no need to compute how the frequentistic program state is affected by the fork and join. If we added the Irrelevant Conditional Rule to the frequency system, then formula (4.8) would become a theorem of the system.

On the other hand, there may be many more sources of incompleteness around. In general, completeness is a trickier property for a system than soundness, because it depends critically upon the exact definition of the formal language in question. Furthermore, the incompleteness that we pointed out in our original Conditional Rule is not all that troublesome. Notice that, if we replace the precondition (4.7) by the very similar but more specific assertion

$$[\mathrm{Fr}(X = \mathrm{H}) = y] \wedge [\mathrm{Fr}(X = \mathrm{T}) = 1 - y], \tag{4.11}$$

the problem evaporates; by our standard Conditional Rule, we can trace y grams of mass through the TRUE branch and $1 - y$ grams through the FALSE branch, and we end up with (4.11) as the output assertion. Thus, the incompleteness of our original Conditional Rule is unlikely to be a major problem for someone attempting to perform the dynamic phase of an algorithmic analysis. Based upon these sorts of experience, we shall generally ignore completeness issues in what follows.

The Loop Rules.

We are left with the task of handling loops. We can get a first approximation to what these rules ought to be by considering the flowcharts of the loops. Working from the flowchart for the **repeat**-loop in Figure 4.2, we deduce that the Repeat Rule should look something like

$$\frac{\vdash [A + B]\ S\ [C],\quad \vdash [C \mid P]\ T\ [B]}{\vdash [A]\ \textbf{loop}\ S\ \textbf{while}\ P\textbf{:}\ T\ \textbf{repeat}\ [C \mid \neg P]}. \tag{4.12}$$

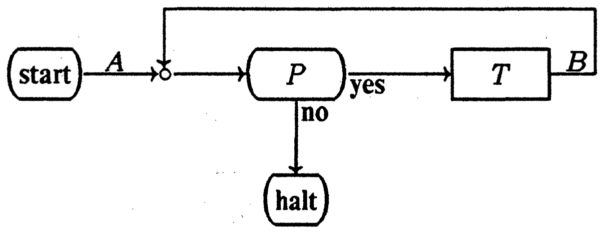

Figure 4.3. A general **while**-loop.

By our convention, the assertion C would be called the summary assertion for the loop. In the special case where S is the empty statement, the **repeat**-loop becomes a **while**-loop; working from Figure 4.3, we have the corresponding intuitive While Rule

$$\frac{\vdash [(A+B) \mid P]\; T\; [B]}{\vdash [A]\ \textbf{while}\ P\ \textbf{do}\ T\ \textbf{od}\ [(A+B) \mid \neg P]}. \tag{4.13}$$

In this case, our convention dictates that the assertion $A+B$ should be called the summary assertion of the **while**-loop.

Rather than dealing with **for**-loops directly, we shall treat the **for**-loop

for J **from** ℓ **to** u **do** S **od**

simply as shorthand for its canonical implementation:

```
$J \leftarrow \ell$;
while $J \le u$ do $S$; $J \leftarrow J+1$ od.
```

These rules have great intuitive appeal; unfortunately, as we saw in Chapter 3, too naive a trust in such intuitions can lead to proofs of false formulas. In particular, we saw that the summary assertion of a loop can describe execution paths other than those that really occur. All realistic execution paths begin at the **start** of the flowchart at some finite time, and either emerge at a **halt** after a finite number of steps, or spend the rest of time looping through the flowchart. But summary assertions, since they have no notion of time or history, can also describe paths that never start and never stop, but spend all of time looping; we called this phenomenon *fictitious mass.* Worse yet, they can describe paths that never start, but do stop, the so-called *time bombs.* Our current task is to determine a collection of restrictions that can be put on the intuitive looping rules to guarantee that the rules are sound. These restrictions will eliminate time bombs; they will not eliminate fictitious mass, but they will guarantee that the effects of its presence are not visible in the input-output behavior of the loop, which is all that the conclusion of a looping rule discusses.

Now that we have a more formal framework in which to operate, we can see that there are several other problems besides time bombs that can lead to unsound conclusions from the intuitive looping rules. First, we must insure that the postassertion of the conclusion of a looping rule describes a subset of $\mathcal{F}^+$ that is closed in an appropriate sense, or we are unable to perform the limiting operation that is inherent in considering all executions of the loop. For example, consider once again the CoinFlip program

$$\textbf{loop } X \leftarrow \text{Random}_{\mathsf{HT}};\ \textbf{while } X = \mathsf{T} \textbf{ repeat.}$$

We noted earlier that the intuitively correct summary assertion for this loop, assuming one gram of input, is

$$[\mathrm{Fr}(X = \mathsf{H}) = 1] \wedge [\mathrm{Fr}(X = \mathsf{T}) = 1].$$

But if we allowed non-vanilla assertions, we could try using the summary assertion

$$[\mathrm{Fr}(X = \mathsf{H}) < 1] \wedge [\mathrm{Fr}(X = \mathsf{T}) < 1].$$

If we follow this assertion once around the loop, we find that it does support itself. In particular, suppose that we start out with h grams of $X = \mathsf{H}$ mass and t grams of $X = \mathsf{T}$ mass, and go once around the loop. The control test sends the h grams $X = \mathsf{H}$ mass out of the loop at once; the other t grams are joined by the one gram of input mass, and this total mass is evenly divided between heads and tails, to leave us in a state in which we have $(1+t)/2$ grams of each kind. More formally, the assertions

$$\begin{aligned} A &= [\mathrm{Fr}(\textsc{true}) = 1] \\ B &= [\mathrm{Fr}(\textsc{true}) < 1] \\ C &= [\mathrm{Fr}(X = \mathsf{H}) < 1] \wedge [\mathrm{Fr}(X = \mathsf{T}) < 1] \end{aligned} \tag{4.14}$$

make the premises of the intuitive Repeat Rule (4.12) true for the CoinFlip program. Unfortunately, the postassertion of the corresponding conclusion,

$$\mathrm{Fr}(X = \mathsf{H}) < 1, \tag{4.15}$$

is not correct. Thus, when assertions are allowed to describe subsets of $\mathcal{F}^+$ that are not appropriately closed, it is possible to get into trouble. This is our motivation for demanding that the postassertion of the conclusion of a loop rule be closed.

The assertions (4.14) cause us to deduce the incorrect postassertion (4.15) for the program CoinFlip. One can explain that bad example from an intuitive point of view by saying that the assertions (4.14) describe everything that happens for any bounded length of time, but, since they specify strict upper bounds on frequencies, they don't allow us to take the limit inherent in considering everything that ever happens. This insight might lead one to expect that we could replace the restriction that assertions be closed with the weaker condition that they contain the limits of bounded increasing sequences. A sequence $\langle c_n \rangle$ in $\mathcal{F}^+$ is called *increasing* if the differences $c_{n+1} - c_n$ are positive measures, that is, are also in $\mathcal{F}^+$; it is called *bounded* if the sequence of real numbers $\langle \|c_n\| \rangle$ is bounded above. If we let c_n describe the mass that exits the loop after performing no more than n iterations of the loop body, we would expect $\langle c_n \rangle$ to be a bounded increasing sequence. Every bounded increasing sequence converges in $\mathcal{F}^+$, and one might guess that it would be sufficient merely to demand that the characteristic sets of our postassertions be closed under bounded increasing sequences.

But that intuition is wrong, and a simple example shows why. Consider the CoinFlip program again, started off with one gram of input mass, and the non-vanilla assertions

$$\begin{aligned} A &= [\mathrm{Fr}(\textsc{true}) = 1] \\ B &= [\mathrm{Fr}(\textsc{true}) > 1] \\ C &= [\mathrm{Fr}(X = \textsc{h}) > 1] \wedge [\mathrm{Fr}(X = \textsc{t}) > 1], \end{aligned} \tag{4.16}$$

which are just the assertions of (4.14) with each "less than" replaced by a "greater than". These assertions also satisfy the premises of the intuitive Repeat Rule. The corresponding postassertion for CoinFlip is

$$\mathrm{Fr}(X = \textsc{h}) > 1, \tag{4.17}$$

which, like (4.15), is incorrect. The characteristic set of (4.17) is not closed, but it is closed under bounded increasing sequences. This example shows that some condition stronger than "closed under bounded increasing sequences" is necessary; we shall stick with the standard concept of closure.

The second failure mode of the intuitive loop rules is a trivial but sweeping point. Consider the intuitive While Rule (4.13), and suppose that the characteristic set of the assertion B is empty. This means that B is equivalent to the assertion FALSE, or, in other terminology, that B is *infeasible.* If the assertion FALSE is conjoined with any other assertion, note that the result will always be FALSE; and note that any restriction of FALSE is also FALSE. Thus, the premise of (4.13) reduces to the formula [FALSE] T [FALSE]; this is a trivially true formula, and, with luck, it will also be a theorem of the frequency system. The intuitive While Rule then allows us to deduce the theorem

$$\vdash [A] \textbf{ while } P \textbf{ do } T \textbf{ od } [\textsc{false}],$$

which is rather a shame because this "theorem" is wrong for any feasible assertion A. A similar problem exists with the intuitive Repeat Rule (4.12) as well, in the case that both B and C are infeasible.

Fortunately, it turns out that this second kind of bad behavior is ruled out by the same restriction that eliminates time bombs. In particular, we shall demand that the summary assertions of loops be feasible assertions. This certainly is not too much to ask, for if the summary assertion is not feasible, it is equivalent to FALSE, and we are in the situation above. The summary assertions that cause time bombs, when viewed from an intuitive point of view, describe frequentistic states with an infinite amount of mass. According to our definitions, however, the characteristic set of an assertion is a subset of $\mathcal{F}^+$, not of $\mathcal{F}^*$; thus, if an assertion does not describe some frequentistic state with finite total mass, it is equivalent to FALSE, regardless of what subset of $\mathcal{F}^*$ it may seem to describe.

We now have enough background to explain the restrictions that turn the intuitive While and Repeat Rules into the real things.

Theorem.

An application of the While Rule

$$\frac{\vdash [(A+B) \mid P]\; T\; [B], \quad \vdash [(A+B) \mid \neg P] \Rightarrow C}{\vdash [A] \textbf{ while } P \textbf{ do } T \textbf{ od } [C]}$$

is guaranteed to be sound if the following two regulations are enforced: the assertion B is feasible, and the assertion C is closed.

Proof.

If the precondition A in the conclusion is infeasible, then the conclusion is vacuously true, and we are done at once. If not, let a denote an arbitrary frequentistic state in $\chi(A)$. The first step in tackling this theorem is to determine what the **while**-loop will in fact do on input a. To determine this, we need to invoke the semantics for the **while**-loop. Let the linear map $f\colon \mathfrak{F} \to \mathfrak{F}$ denote the semantic interpretation of the body T of the loop. Tracing the execution of the loop, we note that the measure $a \mid \neg P$ describes the mass rejected immediately by the control test; mass measured by $a \mid P$ continues in the loop, by going through T. The mass that comes out of the other side of T will be $f(a \mid P)$, by the definition of f. Of this mass, $f(a \mid P) \mid \neg P$ will exit the loop now, having made one trip around. The rest, $f(a \mid P) \mid P$, will start through T for the second time.

Define the function $g\colon \mathfrak{F} \to \mathfrak{F}$ by the relation $g(s) = f(s \mid P)$ for s in $\mathfrak{F}$. The mass that exits the loop after going around exactly n times is given by $g^n(a) \mid \neg P$, where the exponent n denotes the result of composing g with itself n times. We would expect, therefore, that the output of the **while**-loop when started in the input frequentistic state a would be the infinite sum

$$\sum_{n \geq 0} (g^n(a) \mid \neg P). \tag{4.18}$$

This expectation is accurate; we followed Kozen by defining the meaning of the **while**-loop to be the least fixed point of an affine transformation, and Kozen shows that the infinite sum above will converge to that least fixed point [page 16 of 22]. We can thus take the sum of the series (4.18) as the definition of the output of the **while**-loop on input a.

Let c_n denote the nth partial sum of the series (4.18):

$$c_n = \sum_{0 \leq i \leq n} (g^i(a) \mid \neg P).$$

We can show, in fact, that the sequence $\langle c_n \rangle$ forms a bounded increasing sequence in $\mathfrak{F}^+$. Since the initial state a was assumed to be positive, and since the function g takes positive states to positive states, all the terms of the series (4.18) are positive; this shows that the sequence $\langle c_n \rangle$ is increasing. It turns out that the norms $\|c_n\|$ are all bounded by $\|a\|$. On positive states, the norm is a linear functional; hence, for any positive s, we have

$$\|s \mid P\| + \|s \mid \neg P\| = \|s\|.$$

Kozen shows (our Equation (2.1)) that, if f is the semantic interpretation of a program, we have

$$\|f(s)\| \leq \|s\|$$

for any positive state s; this inequality merely states that control cannot exit a program more often than it enters it. Together, these facts demonstrate that

$$\|f(s \mid P)\| + \|s \mid \neg P\| \leq \|s\|,$$

or equivalently,

$$\|g(s)\| + \|s \mid \neg P\| \leq \|s\|.$$

If we iterate this relation by applying it with s repaced by $g(s)$, we can deduce that

$$\left\|g^2(s)\right\| + \|g(s) \mid \neg P\| + \|s \mid \neg P\| \leq \|s\|.$$

Continuing to iterate, we can then deduce in general that

$$\|g^n(s)\| + \sum_{0 \leq i < n} \left\|g^i(s) \mid \neg P\right\| \leq \|s\|. \tag{4.19}$$

Applying this result with s replaced by a, we see that all of the quantities $\|c_n\|$ are bounded by the finite real number $\|a\|$. Therefore, the sequence $\langle c_n \rangle$ is a bounded increasing sequence.

Any bounded increasing sequence in $\mathcal{F}^+$ converges to a limit in $\mathcal{F}^+$; let c_∞ denote the limit of the sequence $\langle c_n \rangle$. This limit c_∞ is the sum of the infinite series (4.18), and is hence our definition of the output of the **while**-loop. If we could somehow demonstrate that all of the measures c_n actually satisfied the output assertion C, and if we knew that the set $\chi(C)$ was closed under bounded increasing sequences, we would be done: the limit of the partial sums c_∞, which defines the output of the **while**-loop, would satisfy C as well. Unfortunately, life is not that easy. We can't begin to deduce anything from the premises of the Rule until we have in our hands some state in $\chi(B)$. This explains why we must assume that the assertion B is feasible.

Since B is feasible, let b be a finite positive measure in $\chi(B)$. From the first premise, we conclude that

$$f(\chi((A+B) \mid P)) \subseteq \chi(B),$$

or, putting it another way,

$$g(\chi(A+B)) \subseteq \chi(B). \tag{4.20}$$

Starting out with a in $\chi(A)$ and b in $\chi(B)$, we can deduce from (4.20) that

$$g(a+b) = g(a) + g(b) \in \chi(B),$$

and thus that

$$g(a) + g(g(a) + g(b)) = g(a) + g^2(a) + g^2(b) \in \chi(B),$$

and thus that

$$g(a) + g(g(a) + g^2(a) + g^2(b)) = g(a) + g^2(a) + g^3(a) + g^3(b) \in \chi(B).$$

In general, we have

$$g^n(b) + \sum_{1 \leq i \leq n} g^i(a) \in \chi(B).$$

If we now add in the state a and then restrict to the falsity of P, we may conclude that, for any n,

$$(g^n(b) \mid \neg P) + \sum_{0 \leq i \leq n} (g^i(a) \mid \neg P) \in \chi((A + B) \mid \neg P).$$

Finally, applying the second premise, we have

$$(g^n(b) \mid \neg P) + \sum_{0 \leq i \leq n} (g^i(a) \mid \neg P) \in \chi(C). \tag{4.21}$$

Although we haven't shown that any of the partial sums of (4.18) lie in $\chi(C)$, we have shown that, if we add to the nth partial sum c_n the correction term ϵ_n given by

$$\epsilon_n = (g^n(b) \mid \neg P),$$

we get a state in $\chi(C)$. Our next goal is to show that these correction terms are small. To see this, note that we can apply inequality (4.19) with s replaced by b to deduce that the partial sums of the series

$$\sum_{n \geq 0} (g^n(b) \mid \neg P) \tag{4.22}$$

also form a bounded increasing sequence. In particular, this guarantees that the norms of the terms must converge to zero, that is, that

$$\|\epsilon_n\| = \|g_n(b) \mid \neg P\| \to 0.$$

The combined sequence $\langle c_n + \epsilon_n \rangle$ must converge, since it is the sum of a bounded increasing sequence and a sequence converging to 0; furthermore, it must converge to the same limit c_∞ to which the sequence $\langle c_n \rangle$ converges. Since the measure $c_n + \epsilon_n$ lies in $\chi(C)$ for all n by (4.21), and since the set $\chi(C)$ is assumed closed, the limit point c_∞ must also lie in $\chi(C)$. But that limit point c_∞ is our definition of the output of the **while**-loop on the input a. Therefore, the output of the **while**-loop on an arbitrary input a satisfying the precondition A is a state that satisfies the postassertion C, and we are done. ∎

Similar theorems will hold for other looping constructs. The general principle involved is the following. An application of an intuitive looping rule is guaranteed to be sound if the following two additional restrictions are enforced: first, the summary assertion of the loop is feasible; and second, the postassertion of the rule's conclusion is closed. In fact, in the first restriction, it is enough if any assertion that cuts the loop is feasible, since we can begin to trace the behavior of the loop from any point. Our choice of which loop-cutting assertion to distinguish as the summary assertion was arbitrary.

From a more practical standpoint, it is important to decide how difficult it is to check the restrictions in this theorem, since they must be formally checked on each invocation of a loop rule. The closure restriction is no problem for us, since we have agreed to limit ourselves to vanilla assertions, and we have already noted that every vanilla assertion is closed. The feasibility restriction is more of a problem. Each time we use a loop rule, we must prove that at least one of the loop-cutting assertions is feasible, that is, is not equivalent to FALSE. As we disussed earlier, the necessary arguments can be quite subtle even for vanilla assertions. The simplest situation pertains in the disjoint vanilla case, when the various predicates in the clauses can be shown to be mutually exclusive, the total of the masses specified by the clauses can be shown to be finite, and none of the clauses ascribes nonzero mass to the predicate FALSE. Any disjoint vanilla assertion is feasible. When dealing with nondiscrete distributions, we shall be unable to limit ourselves to disjoint vanilla assertions, and hence we shall have to find some other way of guaranteeing feasibility.

The frequency system contains Floyd-Hoare verification as a special case. Furthermore, it turns out that the restrictions in this theorem will never present any problem when the derivation being performed is the image of a Floyd-Hoare proof. Recall that the Floyd-Hoare predicate P maps into the assertion $\mathrm{Fr}(\neg P) = 0$. This assertion contains only one clause, with a zero right-hand side. Any such assertion is disjoint vanilla, so that, if we restrict ourselves to making Floyd-Hoare style assertions, we never need to worry about feasibility or closure. In fact, not only are all Floyd-Hoare assertions feasible, they are all satisfied by a single frequentistic state, in particular, the zero state.

We have now completed the description of the frequency system, a sound formal system for reasoning about the probabilistic behavior of programs.

Chapter 5. Using the Frequency System

Getting Answers Out.

We want to perform the dynamic phases of algorithmic analyses inside the frequency system, and this suggests some new questions. First and foremost is the question of how to get information about a performance parameter out of the use of the frequency system. Consider, for example, the program CoinFlip that we discussed earlier:

$$\textbf{loop } X \leftarrow \mathrm{Random}_{\mathrm{HT}}; \textbf{ while } X = \mathrm{T} \textbf{ repeat}.$$

If we start off this program with one gram of input mass, that is, in a state satisfying $\mathrm{Fr}(\textsc{true}) = 1$, we can describe the frequentistic behavior of the loop by the summary assertion

$$\left[\mathrm{Fr}(X = \mathrm{H}) = 1\right] \wedge \left[\mathrm{Fr}(X = \mathrm{T}) = 1\right],$$

which serves to justify the output assertion

$$\left[\mathrm{Fr}(X = \mathrm{H}) = 1\right] \wedge \left[\mathrm{Fr}(X = \mathrm{T}) = 0\right].$$

The output assertion is vanilla, and hence closed; the summary assertion is disjoint vanilla, and hence feasible. Thus, these assertions reflect a sound application of the Repeat Rule. This derivation in the frequency system might be called a *data analysis* of the CoinFlip program, since it discusses just the probabilistic structure of the program's variables. Data analyses are all that we have been considering so far.

An average case algorithmic analysis of the behavior of CoinFlip will focus on the number of flips as the relevant performance parameter; it is the only interesting thing around. There are three possible ways of deducing information about the probability distribution of this performance parameter.

First, we can reason outside the system. From the data analysis of CoinFlip, we can determine the probabilities that the control test will come out each way. In particular, since the summary assertion ascribes equal mass to the events $X = \mathrm{H}$ and $X = \mathrm{T}$, there is probability precisely $\frac{1}{2}$ that control will exit the loop, each time it comes to the control test. This is enough information to allow us to deduce that the coin will be flipped precisely k times with probability exactly 2^{-k} for every positive integer k, and this gives the probability distribution of the performance parameter of interest. From this distribution, we can then compute the average number of flips of the coin, which is of course 2.

This is the technique that Ben Wegbreit used in his paper, and it works satisfactorily. On the other hand, the construction of the frequency system was an attempt to formalize the dynamic phases of algorithmic analysis, and, if we get results about our performance parameters by reasoning outside the frequency system, we are in some sense cheating. Our goal has been to formalize as much as possible of the reasoning of the dynamic phase.

The second possibility that suggests itself allows us to keep more of our reasoning within the system. This second technique is based on the insight that our assertions actually give the weight of execution mass in grams. If we follow the convention that the total weight of the input state should be normalized to be 1 gram, then a mass of g grams indicates that the corresponding event happens on the average g times during one random execution of the program. In particular, applying this insight to CoinFlip, we might conclude that the average number of flips is 2 simply because the summary assertion describes a total of 2 grams of execution mass. In general, to employ this idea, we would attempt to arrange that the assertions in our data analysis all included a clause of the form $\mathrm{Fr}(\text{TRUE}) = e$, either explicitly or implicitly. Then, we would take the value e as the average number of times that control passed the corresponding point in the flowchart.

This total mass technique has an appealing simplicity, but it has several problems. First, it only allows us to determine the average value of our performance parameter, not its distribution in greater detail. This is a minor annoyance, but not too surprising in retrospect; one could hardly get information about the distribution of the number of flips out of the data analysis of CoinFlip, since that analysis doesn't even mention the powers of 2. Fictitious mass, however, presents a second problem that deals a fatal blow to the total mass technique. We have not found any way to eliminate fictitious mass from our summary assertions. Although all the implications of a summary assertion that are visible from outside the loop are in fact correct, the summary assertion may still describe a finite amount of mass that never entered and will never exit the loop, but just goes around and around. Although this fictitious mass does not affect the validity of the theorems that come out of the frequency system analyses, the possibility of its presence harpoons the total mass technique; the value that we get for the average number of executions will simply be wrong if the assertions also happen to describe some fictitious mass.

There is a third technique, however, which operates within the system, gives us distribution information about our parameter beyond its average value, and is also immune to the bad effects of fictitious mass. This is the method of counter variables. We have mentioned the use of counter variables several times already: as a method for performing the upper bound parts of worst case proofs in a Floyd-Hoare system, and as a method for proving termination. They turn out to be an excellent solution to our current problem as well. Suppose that we add to the program a new counter variable C, which is inititalized to zero, and incremented every time the event that we are counting occurs. We shall call the resulting structure a *monitored program.* Since, in the CoinFlip program, we are counting flips, the appropriate monitored program is

```
$C \leftarrow 0$;
**loop** $X \leftarrow \mathrm{Random}_{\mathrm{HT}}$; $C \leftarrow C + 1$; **while** $X = \mathrm{T}$ **repeat**.
```

We can then discuss in our assertions the distribution of the value of C. That is, we can make assertions about the behavior of the monitored program, and verify these assertions by formal manipulations in the frequency system. The output assertion for the monitored program will describe the probability distribution of the performance parameter.

In the CoinFlip example, the truth of the input assertion $\mathrm{Fr}(\textsc{true}) = 1$ of the monitored program implies the truth of the assertion $[\mathrm{Fr}(C = 0) = 1] \wedge [\mathrm{Fr}(C \neq 0) = 0]$ after the intitialization of C. The summary assertion of the **repeat**-loop is

$$[\mathrm{Fr}(C < 1) = 0] \wedge \bigwedge_{c \geq 1} [\mathrm{Fr}(C = c, X = \mathsf{H}) = 2^{-c},\ \mathrm{Fr}(C = c, X = \mathsf{T}) = 2^{-c}]$$

The assertion $\mathrm{Fr}(C < 1) = 0$ with a zero right-hand side simply records the truth of Floyd-Hoare predicate $C \geq 1$. We are assuming that the data type of X allows us to conclude that X must have one of the two values H and T; if this restriction were not built into the data type, we could add the Floyd-Hoare clause $\mathrm{Fr}([X \neq \mathsf{H}] \wedge [X \neq \mathsf{T}]) = 0$ to our summary assertion, and achieve the same effect. Our next task is to follow the mass once around the loop, and we shall elide the Floyd-Hoare assertions in this process. Thus, we shall begin with the summary assertion

$$\bigwedge_{c \geq 1} [\mathrm{Fr}(C = c, X = \mathsf{H}) = 2^{-c},\ \mathrm{Fr}(C = c, X = \mathsf{T}) = 2^{-c}].$$

The mass described by this summary assertion enters the control test, which sends all of the mass with $X \neq \mathsf{T}$ out of the loop. This supports the output assertion

$$\bigwedge_{c \geq 1} [\mathrm{Fr}(C = c, X = \mathsf{H}) = 2^{-c}]$$

for the monitored CoinFlip program. The mass with $X = \mathsf{T}$,

$$\bigwedge_{c \geq 1} [\mathrm{Fr}(C = c, X = \mathsf{T}) = 2^{-c}],$$

is sent around the loop again. At this point, we are no longer interested in the value of X, since it is about to be smashed. Therefore, we might as well throw away that portion of our information. We throw away information by summing; in this case, we sum the frequencies associated with the events $[C = c, X = \mathsf{H}]$ and $[C = c, X = \mathsf{T}]$ to get the frequency of the combined event $[C = c]$. The first of these events has zero frequency (a Floyd-Hoare fact, and hence elided). Thus, the combined event has the same frequency as the second event alone, and we deduce the assertion

$$\bigwedge_{c \geq 1} [\mathrm{Fr}(C = c) = 2^{-c}].$$

The next thing that happens is that the one gram of input mass in which C has just been set to 0 joins the flow; we can reflect this by changing the lower bound on the index c:

$$\bigwedge_{c \geq 0} [\mathrm{Fr}(C = c) = 2^{-c}].$$

Altogether, we now have two grams of mass. The random assignment to X splits every pellet exactly in half, so, coming out of that assignment, we have

$$\bigwedge_{c\geq 0} [\mathrm{Fr}(C = c, X = \mathsf{H}) = 2^{-c-1},\ \mathrm{Fr}(C = c, X = \mathsf{T}) = 2^{-c-1}].$$

To prepare for the deterministic assignment to C, we replace the mathematical variable c by $c - 1$, getting

$$\bigwedge_{c\geq 1} [\mathrm{Fr}(C+1 = c, X = \mathsf{H}) = 2^{-c},\ \mathrm{Fr}(C+1 = c, X = \mathsf{T}) = 2^{-c}].$$

With this assertion on the input to the increment of the counter C, an axiom of assignment will allow us to conclude on output from the increment the assertion

$$\bigwedge_{c\geq 1} [\mathrm{Fr}(C = c, X = \mathsf{H}) = 2^{-c},\ \mathrm{Fr}(C = c, X = \mathsf{T}) = 2^{-c}].$$

And this, neatly enough, is exactly the summary assertion of the loop once again.

What we have done in this example is to use our frequency system techniques on the monitored program, instead of on the original program. The resulting analysis might be called a *performance analysis* in the frequency system, rather than just a data analysis. The output assertion of the monitored program, which, in the CoinFlip case, is

$$[\mathrm{Fr}(C < 1) = 0] \wedge [\mathrm{Fr}(X = \mathsf{T}) = 0] \wedge \bigwedge_{c\geq 1} [\mathrm{Fr}(C = c, X = \mathsf{H}) = 2^{-c}],$$

describes completely the distribution of the performance parameter of interest. The only information from outside the system that we have to apply is our knowledge that the counter variable C will in fact count the number of flips, which is what we set out to study. As the example of CoinFlip demonstrates, the method of counter variables is an excellent way of getting information about a performance parameter out of the frequency system. We shall adopt it exclusively in our other examples.

We performed the above analysis by starting with the correct summary assertion for the loop, and then following it once around the loop to verify its correctness. Because the summary assertion was fairly easy to invent, this was no problem. But, if we didn't realize that the powers of two were involved in the performance analysis of CoinFlip, we could actually discover that fact by reasoning in the frequency system. We could have started our derivation with the summary assertion

$$\bigwedge_{c\geq 1} [\mathrm{Fr}(C = c, X = \mathsf{H}) = p_c,\ \mathrm{Fr}(C = c, X = \mathsf{T}) = q_c], \tag{5.1}$$

where p_c and q_c are unknown parameters whose values we would attempt to discover. As we start to walk this assertion around the loop, the mass in which $X = \mathsf{H}$ exits. The one gram of input mass then joins our flow, and we hence define q_0 to be 1, to avoid making $c = 0$ a special case. The coin flip then splits the flow in half again, and we end up with the assertion

$$\bigwedge_{c \geq 1} \left[\mathrm{Fr}(C = c, X = \mathsf{H}) = \tfrac{1}{2}q_{c-1},\ \mathrm{Fr}(C = c, X = \mathsf{T}) = \tfrac{1}{2}q_{c-1}\right]. \tag{5.2}$$

We can now determine the parameters p_c and q_c so that the two assertions (5.1) and (5.2) are the same. In order for this to be the case, the parameters p_c and q_c must satisfy the identities

$$p_c = \tfrac{1}{2}q_{c-1} \quad \text{and} \quad q_c = \tfrac{1}{2}q_{c-1}, \quad \text{for all } c \geq 1.$$

These identities constitute recurrence relations that, together with the initial condition $q_0 = 1$, allow us to compute that $p_c = q_c = 2^{-c}$. Thus, by formal manipulations in the frequency system, we can deduce the recurrence relations that define the probabilistic behavior of our performance parameter.

Continuous Models.

Having successfully handled the dynamic phase of the average case analysis of CoinFlip, it is time to set our sights a little higher. In particular, our next goal is to tackle the program FindMax, which we considered back in Chapter 1 as a paradigmatic example of the average case analysis of algorithms.

The input to FindMax is a random permutation, and that is the source of some difficulty. Suppose that we in fact let the input array take on as value the $n!$ different permutations on the set $\{1, 2, \ldots, n\}$, each equally likely. Already, we can see part of the difficulty: we would have to describe this probability distribution on the space of all possible values of the array by some sort of frequentistic assertion, and that does not look easy. But even graver problems await. Suppose that the program has examined the first element of the array, and found that it is k. What is a characterization of the randomness that is left in the rest of the array? The remainder of the array will be a random permutation of the set $\{1, 2, \ldots, n\} - \{k\}$. The prospect of describing this in any assertion language that, like ours, deals at the level of the first order predicate calculus is a daunting one. In fact, the same phenomenon arises in program verification research. There are many verification systems that can demonstrate that the output of a sorting program is sorted; but, of those, only a few can also show that the output is some permutation of the input.

It turns out, however, that we can finesse this problem by using a continuous model for a random permutation instead of the discrete model discussed above. Suppose that the elements of the input array are assumed to be independent, identically distributed real random variables. If the distribution from which they are drawn is continuous, the probability of any two of them being equal will be zero. In addition, the elements will be equally likely to be in each of the $n!$ possible orders. Thus, if our algorithm operates by doing comparisons on the values, we can model a random permutation in this continuous manner instead. The advantage of this technique is the tremendous convenience of independence. Not only is it easier to describe the input state, but, if the algorithm has examined the first element of the array, this does not affect in the slightest our state of knowledge about the other elements of the array. They are still independent, identically distributed random variables.

This technique of modeling a random permutation as a sequence of independent, identically distributed random variables works only for those programs that operate exclusively by comparing data values. If an algorithm performs arithmetic on its input, or uses the elements of a permutation as pointers into some other data structure, the continuous model of random permutations will not be a valid substitute for the normal, discrete model. There are many programs in the area of sorting and searching, however, where the continuous model is appropriate, including the program FindMax.

In order to use the continuous model of random permutations, we have to decide upon a continuous probability distribution on the reals. We shall adopt the uniform distribution on $[0, 1)$ as being the most natural. As we commented earlier, we can describe a program variable X that has this distribution by a collection of atomic assertions, each of which specifies the probability that X lies in a particular measurable subset of the real line. The most general such description would be the assertion

$$\bigwedge_{M}[\mathrm{Fr}(X \in M) = \mu(M \cap [0,1))], \tag{5.3}$$

where μ represents Lebesgue measure, and the conjunction ranges over all measurable subsets M of the real numbers.

Rather than working explicitly with conjunctions such as (5.3), it will be convenient to introduce an abbreviated notation, based upon a differential way of viewing things. From an intuitive point of view, the probability that X lies in the differential interval $[x, x + dx)$ is simply dx for x in $[0, 1)$, and 0 elsewhere. We shall adopt the notation $X \approx x$, which might be read "X is differentially equal to x," as an abbreviation for $x \leq X < x + dx$. Then, we can describe a random variable that is uniformly distributed on $[0, 1)$ by the assertion

$$[\mathrm{Fr}(X < 0) = 0] \wedge [\mathrm{Fr}(X \geq 1) = 0] \wedge \bigwedge_{0 \leq x < 1} [\mathrm{Fr}(X \approx x) = dx]. \tag{5.4}$$

We shall often treat this differential type of assertion from an intuitive point of view, and pretend that the various clauses are really giving the differential probabilities associated with X lying in certain differential intervals. Formally, however, such a differential assertion is merely an abbreviation for a conjuntion over all measurable sets. By giving the frequency distribution of a variable X in differential form, we really mean that integrating that differential density over any measurable set M will give the frequency with which X lies in M.

The differential form (5.4) of the assertion (5.3) looks like a vanilla assertion; in fact, we might as well call it vanilla, say *differentially vanilla*, since the assertion (5.3) that it stands for is vanilla. One pleasant property of the differential point of view is that it allows us to invent an analog of the disjoint vanilla property that will be useful for nondiscrete distributions. We shall only suggest the essential concept, without giving details. If the clauses of a differentially vanilla assertion describe the frequencies of events that are mutually exclusive, and if the right-hand sides of the clauses, when summed over discrete indices and integrated over continuous indices as appropriate, add up to a finite number, then that assertion will be feasible. We shall distinguish such assertions by calling them *differentially disjoint vanilla*.

Recall that FindMax goes through its input array from left to right, searching for the maximum element:

```
$M \leftarrow X[1]$;
for $J$ from 2 to $N$ do
    if $X[J] > M$ then $M \leftarrow X[J]$ fi od.
```

We begin to execute this program in a state satisfying $\mathrm{Fr}(N \neq n) = 0$ where n denotes a fixed positive integer. To employ the continuous model of random permutations, we should let the input array $\langle X[1], X[2], \ldots, X[n] \rangle$ consist of n independent random variables, each uniformly distributed on $[0, 1)$. We can characterize this input state by the assertion

$$\begin{aligned} &[\mathrm{Fr}(N \neq n) = 0] \wedge \bigwedge_{1 \leq i \leq n} [\mathrm{Fr}([X[i] < 0] \vee [X[i] \geq 1]) = 0] \\ &\quad \wedge \bigwedge_{\langle x_1, \ldots, x_n \rangle \in [0,1)^n} [\mathrm{Fr}(X[1] \approx x_1, \ldots, X[n] \approx x_n) = dx_1 \ldots dx_n]. \end{aligned}$$

In the particular case of FindMax, however, we can simplify the assertions substantially by using the random assignment feature of our programming language. Since the FindMax program scans through the input sequentially, we can merely generate each random variate on $[0, 1)$ as we need it, rather than generating them all at once, before the program begins. With this technique, we shall have to describe at most two of the random variables at any time: the current maximum and the new challenger.

Our next step, then, is to recode the FindMax program in accord with the incremental approach to generating the random input. The resulting code is

```
$M \leftarrow \mathrm{Random}_U$;
for $J$ from 2 to $N$ do
    $T \leftarrow \mathrm{Random}_U$;
    if $T > M$ then $M \leftarrow T$ fi od;
```

where U represents the uniform distribution on $[0, 1)$. We can now begin to get some sense of what the summary assertion for the **for**-loop will look like. Some clauses will presumably describe the distribution of M, the current maximum. The maximum of k independent uniform variates on $[0,1)$ has the density $d(x^k) = kx^{k-1}\,dx$; therefore, adopting differential format, we might see clauses of the form

$$\bigwedge_{\substack{0\le m<1\\ 2\le j\le n+1}} \bigl[\mathrm{Fr}(M \approx m, J = j) = (j-1)m^{j-2}\,dm\bigr].$$

Recall that the summary assertion of a **for**-loop discusses all the mass entering the control test of the loop, in terms of the *incremented* value of the loop variable. Thus, at the moment that we make our summary assertion, the current maximum M is the maximum of $J-1$ rather than J values. In particular, this means that the mass entering the **for**-loop for the first time is described in the summary assertion as having J equal 2; in this mass, the event $M \approx m$ is ascribed $1m^0\,dm = dm$ grams, which is correct.

But we want to do a performance analysis of FindMax, not merely a data analysis. Therefore, our next step is to add a counter variable C, which will keep track of the number of left-to-right maxima, the performance parameter of interest. The monitored version of FindMax is

```
$C \leftarrow 0$; $M \leftarrow \mathrm{Random}_U$;
for $J$ from 2 to $N$ do
    $T \leftarrow \mathrm{Random}_U$;
    if $T > M$ then $M \leftarrow T$; $C \leftarrow C + 1$ fi od.
```

We shall now add to our summary assertion some clauses that discuss the probabilistic behavior of C. Since this probabilistic behavior is what we are trying to determine, we shall simply describe it by a sequence of unknown parameters. In particular, we can add the following clauses to our growing summary assertion:

$$\bigwedge_{\substack{c\ge 0\\ 2\le j\le n+1}} \bigl[\mathrm{Fr}(C = c, J = j) = p_{c,j}\bigr].$$

The coefficients $p_{c,j}$ are intuitively the probabilities that a random permutation on $j-1$ elements has precisely c left-to-right maxima, where the leftmost element is not counted as a maximum.

Even with clauses that describe the distributions both of M and of C, our summary assertion still is not complete. The problem is that our current clauses only discuss the marginal distributions of M and C; not a word is said about their joint distribution. The critical fact about the FindMax program, the thing that makes it pleasant to analyze, is that M and C are independent. To see this, note that M, the current maximum, depends only upon the set of values that have been seen so far; while C, the number of left-to-right maxima seen so far, depends only upon the order in which those values were seen. This independence allows us to combine the clauses that describe M and C into the true summary assertion

$$\bigwedge_{\substack{c\ge 0\\ 0\le m<1\\ 2\le j\le n+1}} \bigl[\mathrm{Fr}(M \approx m, C = c, J = j) = p_{c,j}(j-1)m^{j-2}\,dm\bigr]. \tag{5.5}$$

Note that this summary assertion (5.5) is differentially disjoint vanilla, so that it is also feasible. Our basic job is to carry (5.5) once around the loop; we hope to end up with a new summary assertion that can be matched against (5.5) by choosing appropriate values for the coefficients $p_{c,j}$. Of course some of the mass described by (5.5) will exit the loop on this iteration; and some new mass will enter for the first time. We hope that these effects will balance out. We would expect that the $j = 2$ portion of (5.5) would be supported not by mass coming around from the end of the loop body, but rather by the original input mass. We shall first explore this expectation.

We enter the FindMax program with one gram of mass in which N has the value n. After the first two assignments, the mass entering the **for**-loop is

$$\bigwedge_{0 \leq m < 1} \left[\mathrm{Fr}(M \approx m, C = 0) = dm \right]. \tag{5.6}$$

Note that the relations $0 \leq M < 1$ and $N = n$ are Floyd-Hoare facts at this point; they are specified by assertions with zero right-hand sides that we are choosing to elide. In particular, we could add the conjunct $N = n$ to the predicate in these atomic assertions without changing anything.

Just before the input flow (5.6) to the **for**-loop joins the flow already in the loop, there is an implicit assignment of 2 to J; thus, the input flow at this join is described by

$$\bigwedge_{0 \leq m < 1} \left[\mathrm{Fr}(M \approx m, C = 0, J = 2) = dm \right]. \tag{5.7}$$

In order for this mass to take care of the $j = 2$ portion of our summary assertion (5.5), we only need to guarantee that the coefficient $p_{c,2}$ is 1 if $c = 0$ and 0 otherwise. The Kronecker delta function δ_{ij} is 1 if $i = j$ and 0 otherwise; thus, we want the coefficients $p_{c,j}$ to satisfy the initial condition $p_{c,2} = \delta_{c0}$.

We shall now begin at the summary assertion (5.5), and move once around the loop. We hope to support the $j > 2$ portion of the summary assertion with the result of this round trip. The first thing that we come across is the control test, which compares J and N. This test will cause all of the mass in which $J > N$, or equivalently, in which $j > n$, to exit the loop; the only mass with $j > n$ is that with $j = n + 1$, and it generates the output assertion

$$\bigwedge_{\substack{c \geq 0 \\ 0 \leq m < 1}} \left[\mathrm{Fr}(C = c, M \approx m, J = n + 1) = p_{c,n+1}\, n m^{n-1}\, dm \right].$$

This output assertion is differentially vanilla, and hence closed.

The rest of the mass,

$$\bigwedge_{\substack{c \geq 0 \\ 0 \leq m < 1 \\ 2 \leq j \leq n}} \left[\mathrm{Fr}(M \approx m, C = c, J = j) = p_{c,j}(j - 1) m^{j-2}\, dm \right],$$

stays in the loop. The first action of the loop body is the assignment to T of a uniform random variate on $[0,1)$. This is reflected in the state, according to an axiom of random choice, by changing the assertion to

$$\bigwedge_{\substack{c \geq 0 \\ 0 \leq m < 1 \\ 2 \leq j \leq n \\ 0 \leq t < 1}} \left[\operatorname{Fr}(T \approx t, M \approx m, C = c, J = j) = p_{c,j}(j-1)m^{j-2}\,dm\,dt\right].$$

We can simply multiply the old right-hand side by dt when we add the conjunct $T \approx t$ because the randomly chosen value for T is assumed to be independent of everything that happened previously.

This assertion now arrives at the **if**-test, which compares the values of M and T, or equivalently, of m and t. We shall follow the FALSE branch of the **if**-statement first. The mass that begins this branch is described by the assertion

$$\bigwedge_{\substack{c \geq 0 \\ 2 \leq j \leq n \\ 0 \leq t \leq m < 1}} \left[\operatorname{Fr}(T \approx t, M \approx m, C = c, J = j) = p_{c,j}(j-1)m^{j-2}\,dm\,dt\right]. \qquad (5.8)$$

Note that a Floyd-Hoare system could show that $T \leq M$ on the FALSE branch of the **if**-test. With this in mind, we have elided the atomic assertion $\operatorname{Fr}(T > M) = 0$ in assertion (5.8). (This atomic assertion is written with an inequality, but it is of course vanilla; the inequality is at the predicate level rather than at the assertion level.)

The FALSE branch of the **if**-statement has no executable code; thus, we can put aside assertion (5.8) until we want to recombine the flows at the end of the **if**-statement. Before we put it aside, however, it would be a good idea to throw away the information in it about the value of T. If the latest element was not a left-to-right maximum, we have no further interest in its value, and we would rather that it didn't hang around and clutter our assertion. We want to replace our data about the joint distribution of T and M with data about the marginal distribution of M alone. To do this, we want to sum over all possible values of T; since T is a continuous variable, this really means to integrate over all t. In assertion (5.8), the only nonzero mass is associated with those cases where t lies between 0 and m. Thus, the integral involved is

$$\int_0^m \left(p_{c,j}(j-1)m^{j-2}\,dm\right)dt = p_{c,j}(j-1)m^{j-1}\,dm.$$

The resulting assertion for the end of the FALSE branch that does not discuss T is therefore

$$\bigwedge_{\substack{c \geq 0 \\ 2 \leq j \leq n \\ 0 \leq m < 1}} \left[\operatorname{Fr}(M \approx m, C = c, J = j) = p_{c,j}(j-1)m^{j-1}\,dm\right]. \qquad (5.9)$$

We next go back and consider what has been happening on the TRUE branch of the if-statement. Starting off down the TRUE branch, we have mass described by

$$\bigwedge_{\substack{c\geq 0\\ 2\leq j\leq n\\ 0\leq m<t<1}} \left[\mathrm{Fr}(T\approx t, M\approx m, C=c, J=j)=p_{c,j}(j-1)m^{j-2}\,dm\,dt\right], \tag{5.10}$$

where we have elided the Floyd-Hoare assertion $\mathrm{Fr}(T\leq M)=0$. The first thing that happens on the TRUE branch is the assignment $M\leftarrow T$. That is, since T represents a new left-to-right maximum, we want to record its value in M. In particular, note that the current value of M is no longer of any use to us. Our first action, then, is to integrate out our information about M, that is, to integrate over all values of m. The integral involved is

$$\int_0^t \left(p_{c,j}(j-1)m^{j-2}\,dt\right)dm = p_{c,j}t^{j-1}\,dt,$$

since $j\geq 2$. Thus, a coarser description of the mass beginning the TRUE branch is

$$\bigwedge_{\substack{c\geq 0\\ 2\leq j\leq n\\ 0\leq t<1}} \left[\mathrm{Fr}(T\approx t, C=c, J=j)=p_{c,j}t^{j-1}\,dt\right].$$

To prepare for the future, we shall replace the mathematical variable t by m, and the mathematical variable c by $c-1$. The resulting equivalent assertion is

$$\bigwedge_{\substack{c\geq 1\\ 2\leq j\leq n\\ 0\leq m<1}} \left[\mathrm{Fr}(T\approx m, C+1=c, J=j)=p_{c-1,j}m^{j-1}\,dm\right].$$

From this assertion, appropriate assignment axioms will show that we may assert

$$\bigwedge_{\substack{c\geq 1\\ 2\leq j\leq n\\ 0\leq m<1}} \left[\mathrm{Fr}(M\approx m, C=c, J=j)=p_{c-1,j}m^{j-1}\,dm\right] \tag{5.11}$$

at the end of the TRUE branch.

The last action of the if-statement is to combine the masses that emerge from the TRUE and FALSE branches; we can trace this by adding assertions (5.9) and (5.11). In order to do so, however, we must readjust the lower limit on the index c in (5.11) from 1 to 0. This readjustment will make no difference if we can arrange that the coefficients $p_{c,j}$ satisfy the condition $p_{c,j}=0$ for $c<0$. With this convention, the sum of (5.9) and (5.11) is

$$\bigwedge_{\substack{c\geq 0\\ 2\leq j\leq n\\ 0\leq m<1}} \left[\mathrm{Fr}(M\approx m, C=c, J=j)=\left((j-1)p_{c,j}+p_{c-1,j}\right)m^{j-1}\,dm\right]. \tag{5.12}$$

Since we have now completed the body of the loop, our next task is to deal with the index variable J. To prepare for the incrementation of J, we can replace j by $j-1$ in (5.12), getting

$$\bigwedge_{\substack{c\geq 0\\ 0\leq m<1\\ 3\leq j\leq n+1}} \left[\mathrm{Fr}(M\approx m, C=c, J+1=j) = \left((j-2)p_{c,j-1}+p_{c-1,j-1}\right)m^{j-2}\,dm\right],$$

which the incrementation of J then changes into

$$\bigwedge_{\substack{c\geq 0\\ 0\leq m<1\\ 3\leq j\leq n+1}} \left[\mathrm{Fr}(M\approx m, C=c, J=j) = \left((j-2)p_{c,j-1}+p_{c-1,j-1}\right)m^{j-2}\,dm\right]. \tag{5.13}$$

Our goal is to make the mass described by (5.13) support all of the $j>2$ portion of summary assertion (5.5); the $j=2$ portion has already been handled by the input mass (5.7). Comparing assertions (5.13) and (5.5), we find that we can achieve our goal by guaranteeing that the coefficients $p_{c,j}$ satisfy the identity

$$(j-1)p_{c,j} = (j-2)p_{c,j-1}+p_{c-1,j-1}.$$

Therefore, we shall define the coefficients $p_{c,j}$ for $c\geq 0$ and $j>2$ by the recurrence relation

$$p_{c,j} = \frac{j-2}{j-1}p_{c,j-1}+\frac{1}{j-1}p_{c-1,j-1}$$

under the initial conditions $p_{c,2}=\delta_{c0}$ and $p_{c,j}=0$ for $c<0$. They will then specify the probabilistic behavior of the number of left-to-right maxima in the sense that the assertion

$$\bigwedge_{\substack{c\geq 0\\ 0\leq m<1}} \left[\mathrm{Fr}(C=c, M\approx m, J=n+1) = p_{c,n+1}\,nm^{n-1}\,dm\right]$$

will describe the output of the FindMax program. If we are only interested in the probabilistic structure of C, as opposed to its joint distribution with M, we can integrate m out of this output assertion. Since

$$\int_0^1 nm^{n-1}dm = 1,$$

we deduce that the marginal distribution of C is given by

$$\bigwedge_{c\geq 0}\left[\mathrm{Fr}(C=c)=p_{c,n+1}\right].$$

This justifies our earlier intuitive definition of the $p_{c,j}$ as the probability of c left-to-right maxima other than the leftmost element, in a permutation of length $j-1$.

The FindMax example provides some real evidence that the motivating ideas behind our project were basically sound. By manipulations in a formal system, we have been able to verify that the distribution of the chosen performance parameter is specified by a certain recurrence relation. That is, we have formalized the dynamic phase of the analysis of FindMax, without appealing to poorly undertsood "intuition" about the program. The recurrence relation came naturally out of a flow analysis of the **for**-loop. With this recurrence relation in hand, we could then move on to the static phase of the analysis, where the solution to the recurrence is studied [Section 1.2.10 in 18].

FindMax with Arbitrary Distributions.

Our performance analysis of FindMax depended critically for its success upon the fact that the distribution from which the elements were drawn was assumed continuous. We actually assumed that the data elements were drawn from the uniform distribution on $[0,1)$, but that was not critical; any continuous distribution would have done as well. On the other hand, if the data elements are drawn from a distribution that contains mass points, then one of the basic assumptions of the analysis above is violated: the values of C and M are no longer independent. As an extreme case, consider the distribution that assigns probability $\frac{1}{2}$ to each of 0 and 1. Suppose that we have drawn a sequence of independent variates from this distribution, and that the maximum M of that sequence so far is 0. Note that this implies that C must also be 0. But if the maximum of the sequence were 1, then C could be either 0 or 1, depending upon whether the first element of the sequence was or was not 1. This implies that M and C are dependent. It is still true that M depends only upon the set of values seen so far, but C depends not only on the order in which those values were seen, but also upon the duplicate structure. And the value of the current maximum can give us some indication of the number of duplicates.

In this section, we shall consider how far we can get in a performance analysis of FindMax if we drop the continuity assumption on the distribution. Let $F\colon \mathbf{R} \to \mathbf{R}$ be a nondecreasing function. At a point y of discontinuity of F, the value $F(y)$ of F itself is not as important as the values that F approaches in the limit as we move towards y from above and below. We shall denote these limits by

$$F^{+}(y) = \lim_{x \downarrow y} F(x) \qquad \text{and} \qquad F^{-}(y) = \lim_{x \uparrow y} F(x).$$

There is associated with F a positive measure on the Borel sets that assigns to each open interval (a,b) the measure $F^{-}(b) - F^{+}(a)$ and to each closed interval $[a,b]$ the measure $F^{+}(b) - F^{-}(a)$ [10]. The measure associated with F will be a probability measure if the difference

$$\lim_{x \to \infty} F(x) - \lim_{x \to -\infty} F(x)$$

is 1, and it will have finite total mass if this difference is finite. We shall refer to the measure associated with F by dF, in a Lebesgue-Stieltjes style.

Suppose that the real-valued program variable X has the frequency distribution described by such a measure dF with finite total mass. We shall describe the variable X by the differential assertion

$$\bigwedge_{x} \bigl[\mathrm{Fr}(X \approx x) = dF(x)\bigr]. \tag{5.14}$$

In the past, we used differential assertions only in cases where the distributions involved were differentiable, so that the right-hand sides were actually densities. But remember that a differential assertion is only a shorthand for a conjunction over all measurable sets; the differential assertion (5.14), for example, expands into the conjunction

$$\bigwedge_{M}[\mathrm{Fr}(X \in M) = dF(M)],$$

where the right-hand sides here represent the mass ascribed by the measure dF to the measurable set M.

Thus, the differential format for assertions makes sense when the associated cumulative distribution functions are not differentiable, and even when they are not continuous. The concept of an assertion being differentially disjoint vanilla also makes sense in this more general environment, since there exist random variables that have an arbitrary nondecreasing function F as their cumulative distribution function. If we allow arbitrary distributions, however, we should make one tactical retreat. The measures dF associated with nondecreasing functions F are defined on the Borel sets, rather then on all Lebesgue measurable sets; therefore, we hereby change our conventions by adopting the Borel sets as the natural σ-algebra for the real numbers.

In our previous analysis of FindMax, the random numbers were chosen with the uniform distribution U on $[0, 1)$. With our new notation, the old summary assertion (5.5) could be rewritten

$$\bigwedge_{\substack{c \geq 0,\ m \\ 2 \leq j \leq n+1}} [\mathrm{Fr}(M \approx m, C = c, J = j) = p_{c,j}\, dU^{j-1}(m)],$$

since the maximum of $j-1$ independent U-distributed random variables will be U^{j-1}-distributed.

Suppose that the random numbers input to FindMax are chosen with an arbitrary probability distribution F, rather than the uniform distribution U. Since the values of M and C are not guaranteed to be independent, we shall have to restructure our summary assertion; in particular, we shall no longer be able to factor the right-hand side into the product of a function of c and a function of m. One possibility is to define a two parameter family of measures $dG_{c,j}$, where the measure $dG_{c,j}$ will describe the frequency distribution of M over all of the mass with $C = c$ and $J = j$. Since the total frequency associated with a particular value of C and J will be less than 1, the measures $dG_{c,j}$ will not be probability measures; instead, we might call them *frequency measures*. The analog to the numbers $p_{c,j}$ in this new version of the problem will be the norm of the measures $dG_{c,j}$, the result of integrating $dG_{c,j}(m)$ over all m. In particular, the expression

$$\int_{m} dG_{c,j}(m)$$

will give the probability of precisely c left-to-right maxima other than the leftmost in a sequence of $j-1$ numbers drawn independently from the distribution F.

This suggests that we choose

$$\bigwedge_{\substack{c \geq 0,\ m \\ 2 \leq j \leq n+1}} [\mathrm{Fr}(M \approx m, C = c, J = j) = dG_{c,j}(m)] \tag{5.15}$$

as our new summary assertion for the **for**-loop (this assertion is differentially disjoint vanilla, and hence feasible). What do we find when we carry this assertion once around the loop? The mass with $j = n + 1$ exits the loop; then, the variable T is assigned a random variate with distribution F, resulting in the state

$$\bigwedge_{\substack{c\geq 0, m, t\\ 2\leq j\leq n}} \left[\mathrm{Fr}(T \approx t, M \approx m, C = c, J = j) = dF(t)\, dG_{c,j}(m)\right].$$

This mass then splits at the **if**-test, and the TRUE branch is subjected to various assignments. Tracing things through, it turns out that we shall get back to summary assertion (5.15) again if the measures satisfy the identity

$$dG_{c,j}(m) \;=\; dG_{c,j-1}(m) \int_{t\leq m} dF(t) \;+\; dF(m) \int_{t<m} dG_{c-1,j-1}(t),$$

or equivalently,

$$dG_{c,j}(m) \;=\; F^{+}(m)\, dG_{c,j-1}(m) \;+\; G^{-}_{c-1,j-1}(m)\, dF(m)$$

for $c \geq 0$ and $j \geq 2$. Although this relation looks like a differential equation, it should be considered instead as a recurrence on the associated measures. The appropriate initial conditions are $dG_{c,2}(m) = \delta_{c0}\, dF(m)$ and $dG_{c,j}(m) = 0$ for $c < 0$.

For any particular distribution function F, we can carry out the above calculations, and determine the probabilistic structure of the number of left-to-right maxima. Certain aspects of this computation can be completed even in the very general framework above. For example, we can compute all of the distributions $G_{0,j}$ quite easily; iterating the recurrence, we find for $j \geq 2$ that

$$dG_{0,j}(m) = \left(F^{+}(m)\right)^{j-2} dF(m).$$

Intuitively, the left-hand side gives the differential frequency with which the maximum of a sequence of $j - 1$ variates is m while the sequence has no left-to-right maxima other than its leftmost element. The right-hand side gives the differential frequency $dF(m)$ with which the leftmost element will be m, times the probabilities $F^{+}(m)$ that each of the other $j - 2$ elements will turn out to be no larger than m. Unfortunately, for all but the simplest non-continuous distributions F, the resulting measures $dG_{c,j}$ do not seem to have a simple general form [exercise 1.2.10–10 in 18].

Analyzing a Trivial Algorithm.

We mentioned in Chapter 1 the paper *A Trivial Algorithm whose Analysis Isn't* by Jonassen and Knuth [16]. This paper discusses the structure of binary search trees of sizes 2 and 3 under repeated random insertions and deletions. The analysis is rather subtle, and its dynamic phase serves as a good example of an analysis at a very low level. Jonassen and Knuth derive integral equations for the relevant performance parameter by reasoning almost directly from the code of the program. In this section, we shall show how this dynamic phase looks when couched in the frequency system.

The process to be studied is the following: take an empty binary search tree. Choose one key at random, and insert it, and then repeat this operation. The tree will now have one of two possible shapes. Choose yet another key at random, and insert it; the tree will now have one of five possible shapes, each with an associated probability. Next, choose one of the three keys in the tree, each key chosen equally likely, and delete it using Hibbard's deletion algorithm. Once again, choose a new key at random, and insert it; then choose one of the three current keys at random and delete it. Repeat this insertion-deletion loop indefinitely. The problem is to study the probability distribution of the shapes of the resulting trees, as a function of the number of insert-delete steps. This particular regimen of insertions and deletions is interesting because it constitutes a simple but non-trivial case. The subtlety of the problem comes from the interplay between the probability distribution of the shape of the tree and the probability distributions of the various keys in the tree.

There are only two possible shapes that a binary search tree with two nodes can have, and only five possible shapes for a binary search tree on three nodes. Jonassen and Knuth distinguish between these shapes by means of a letter code, and we shall use the same notation. Thus, the two possible shapes for a tree with two keys are called F and G, while the five possible three-key trees are called A, B, C, D, and E. We shall use the program variable S to hold the shape of the tree when it contains two keys, and T to hold the shape of the three-key trees. The keys of a two-key tree will be stored in the variables V and W in such a way that $V < W$, while the keys of a three-key tree will be stored in X, Y, and Z in the order $X < Y < Z$.

It would be possible to tackle this problem with a discrete model of randomness. For the first n insert-delete cycles, there would be $(n+3)!3^n$ different equally likely possibilities: $(n+3)!$ orders for the $n+3$ inserted keys, and 3 choices at each of the n deletions. But Jonassen and Knuth warn us that "Such a discrete approach leads to great complications." Instead, this is a perfect opportunity to employ a continuous model; the description of the process given above is much closer, in fact, to an approach based upon successive random choices from a continuous key space. Therefore, we shall choose our keys to be independent random variables from a fixed distribution on the real numbers; any continuous distribution would do, but again we shall adopt the uniform distribution U on $[0,1)$ as being the simplest. The choices of which key to delete will be effected by choosing independent random variables from another distribution, one that assigns probability $\frac{1}{3}$ to each of the three outcomes X, Y, and Z. We shall call this distribution XYZ.

We can now write down our first approximation to the InsertDelete program. If we take as input to the program the result of the first two random insertions, the program will have the form

```
while TRUE do
    $R \leftarrow \text{Random}_U$;
    $\langle T; X, Y, Z\rangle \leftarrow \text{Insert}(R, S, V, W)$;
    $L \leftarrow \text{Random}_{XYZ}$;
    $\langle S; V, W\rangle \leftarrow \text{Delete}(L, T, X, Y, Z)$;
od.
```

The functions Insert and Delete produce vectors as output, which are assigned component by component to the vectors on the left of the assignment. We shall refine these functions shortly into **case**-statements that incorporate in a table the rules for all of the possible insertions and deletions in our trees. But first we should devote some effort to roughing out the structure of our frequentistic assertions.

We run into one problem right away: the program InsertDelete as given above never halts, and thus there is an infinite amount of interesting mass going around the **while**-loop. The frequency system doesn't allow us to discuss an infinite amount of mass, and the technique of tacit divergence won't help, since we want to study all of this mass in detail. The best thing to do is to replace the **while**-loop by a **for**-loop that performs n insert-delete cycles for some mathematical variable n. The loop in this modified program will have only n grams of mass running around it (or $n+1$, depending upon where one counts), and the output assertion will discuss the probabilistic state after the nth insert-delete cycle. It is interesting to note that, to be rigorous, we would have had to adopt the **for**-loop modification even if there were no infinite mass restriction in the frequency system. Remember that, because of the possibility of fictitious mass, the frequency system only certifies as accurate the input-output performance of the analyzed program. It is tempting to think that, as in the inductive assertion method, all of the assertions throughout the program will correctly describe the mass going by the corresponding points, but there is no guarantee of this. Of course, if we attempt to analyze the **while**-loop version of the InsertDelete program, the output assertion will be $\text{Fr}(\text{TRUE}) = 0$. Since only that output assertion is trustworthy, all that we are able to conclude is that the **while**-loop version of InsertDelete never terminates (or at least terminates only with probability zero).

We are trying to determine the distributions of the tree shapes, and we find that, to do so, we must in fact keep track of the joint distributions of the shapes and the keys. But we don't have to add any counter variables to the program in this case; a data analysis alone will tell us what we want to know. By changing to a **for**-loop with index variable K and refining the Insert and Delete functions into **case**-statements, we arrive at a version of the program InsertDelete that is tuned for the frequency system:

```
$[\![\alpha]\!]$ for $K$ from 1 to $n$
  $[\![\sigma]\!]$ do
    $[\![\beta]\!]$ $R \leftarrow \text{Random}_U$;
    $[\![\gamma]\!]$ case
          $S = \mathsf{F},\ 0 < R < V \Rightarrow [\![\delta]\!](\langle T; X, Y, Z\rangle \leftarrow \langle \mathsf{A}; R, V, W\rangle)[\![\epsilon]\!]$
          $S = \mathsf{F},\ V < R < W \Rightarrow (\langle T; X, Y, Z\rangle \leftarrow \langle \mathsf{B}; V, R, W\rangle)$
          $S = \mathsf{F},\ W < R < 1 \Rightarrow (\langle T; X, Y, Z\rangle \leftarrow \langle \mathsf{C}; V, W, R\rangle)$
          $S = \mathsf{G},\ 0 < R < V \Rightarrow (\langle T; X, Y, Z\rangle \leftarrow \langle \mathsf{C}; R, V, W\rangle)$
          $S = \mathsf{G},\ V < R < W \Rightarrow (\langle T; X, Y, Z\rangle \leftarrow \langle \mathsf{D}; V, R, W\rangle)$
          $S = \mathsf{G},\ W < R < 1 \Rightarrow (\langle T; X, Y, Z\rangle \leftarrow \langle \mathsf{E}; V, W, R\rangle)$
      endcase;
    $[\![\zeta]\!]$ $L \leftarrow \text{Random}_{\mathsf{XYZ}}$;
    $[\![\eta]\!]$ case
          $T = \mathsf{A},\ L = \mathsf{X} \Rightarrow [\![\theta]\!](\langle S; V, W\rangle \leftarrow \langle \mathsf{F}; Y, Z\rangle)[\![\iota]\!]$
          $T = \mathsf{A},\ L = \mathsf{Y} \Rightarrow (\langle S; V, W\rangle \leftarrow \langle \mathsf{F}; X, Z\rangle)$
          $T = \mathsf{A},\ L = \mathsf{Z} \Rightarrow (\langle S; V, W\rangle \leftarrow \langle \mathsf{F}; X, Y\rangle)$
                    ...cases for $T \in \{\mathsf{B}, \mathsf{C}, \mathsf{D}\}$...
          $T = \mathsf{E},\ L = \mathsf{X} \Rightarrow (\langle S; V, W\rangle \leftarrow \langle \mathsf{G}; Y, Z\rangle)$
          $T = \mathsf{E},\ L = \mathsf{Y} \Rightarrow (\langle S; V, W\rangle \leftarrow \langle \mathsf{G}; X, Z\rangle)$
          $T = \mathsf{E},\ L = \mathsf{Z} \Rightarrow (\langle S; V, W\rangle \leftarrow \langle \mathsf{G}; X, Y\rangle)$
      endcase; $[\![\kappa]\!]$
  od $[\![\omega]\!]$.
```

There are several fine points. First, it is mildly illegal to refer to a mathematical variable in program text (unless perhaps that variable is considered to be a compile-time constant). Thus, we should really make the **for**-loop run from 1 to N for some program variable N, and then agree to enter InsertDelete in a state in which the relation $N = n$ is a Floyd-Hoare fact. We shall stay with the current, mildly illegal version for simplicity. Secondly, the conditions on the arms of the first **case**-statement are not really exhaustive. However, they are exhaustive except for a set of frequency zero, and that is enough. Thirdly, we shall elide in what follows the assertions of many Floyd-Hoare facts, including the relations $S \in \{\mathsf{F}, \mathsf{G}\}$ and $T \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}$ among others.

The atomic assertions that we make should give the differential frequency associated with a particular combination of values of K, S, V, and W when the tree has two keys, and combination of values of K, T, X, Y, and Z when it has three. We shall use a special purpose nomenclature for the assertions that we shall be generating. Assertions that describe the values of K, S, V, and W will be called S-assertions, while those that describe K, T, X, Y, and Z are T-assertions. The other component of an assertion's name is the control point at which it applies. The twelve control points are labelled in the program above; in particular, note that σ is the point where the summary assertion applies, and that α and ω are the input and output points respectively.

We shall start off with a summary assertion that has both an S-assertion part and a T-assertion part, with appropriate unknown coefficients as the right-hand sides. The S-assertion part is the assertion $(\sigma.S)$, given by

$$\bigwedge_{\substack{1\le k\le n+1\\ s\in\{\mathsf{F},\mathsf{G}\}\\ 0<v<w<1}} \bigl[\mathrm{Fr}(K=k, S=s, V\approx v, W\approx w) = P_k(s;v,w)\,dv\,dw\bigr], \tag{$\sigma.S$}$$

while the T-assertion part is

$$\bigwedge_{\substack{2\le k\le n+1\\ t\in\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E}\}\\ 0<x<y<z<1}} \bigl[\mathrm{Fr}(K=k, T=t, X\approx x, Y\approx y, Z\approx z) = Q_k(t;x,y,z)\,dx\,dy\,dz\bigr]. \tag{$\sigma.T$}$$

The functions P_k and Q_k describe the frequentistic structure of the trees of sizes two and three respectively. We have started the index k in assertion $(\sigma.T)$ at 2 instead of at 1, because, on input to InsertDelete, only the variables associated with a tree of size two will have a well-defined meaning. Since the variables T, X, Y, and Z are not defined when $K=1$, we simply won't describe them. Since our keys are being chosen from a continuous distribution, coincidences in which two keys happen to be equal or a key happens to be exactly 0 or 1 occur only with frequency zero; we can ignore them.

Note that our summary assertion is not differentially disjoint vanilla; the S and T halves have this property when considered separately, but their conjunction does not. A different loop-cutting assertion will be differentially disjoint vanilla, however, and recall that it is enough if we can show that any loop-cutting assertion is feasible.

There is a straightforward correspondence between the P_k and Q_k functions and the differential probabilities a_n, b_n, ..., f_n that are used by Jonassen and Knuth. The detailed translation is given by the following table of relations:

$$\begin{aligned}
P_{k+1}(\mathsf{F};v,w) &= f_k(v,w)\\
P_{k+1}(\mathsf{G};v,w) &= g_k(v,w)\\
Q_{k+2}(\mathsf{A};x,y,z) &= a_k(x,y,z)\\
Q_{k+2}(\mathsf{B};x,y,z) &= b_k(x,y,z)\\
Q_{k+2}(\mathsf{C};x,y,z) &= c_k(x,y,z)\\
Q_{k+2}(\mathsf{D};x,y,z) &= d_k(x,y,z)\\
Q_{k+2}(\mathsf{E};x,y,z) &= e_k(x,y,z).
\end{aligned}$$

By carrying our summary assertion once around the loop, we shall determine recurrences that define the functions P_k and Q_k. The summary assertion first undergoes the control test of the **for**-loop, which sends mass out of the loop, described by the conjunction of

$$\bigwedge_{\substack{s\in\{\mathsf{F},\mathsf{G}\}\\ 0<v<w<1}} \bigl[\mathrm{Fr}(S=s, V\approx v, W\approx w) = P_{n+1}(s;v,w)\,dv\,dw\bigr] \tag{$\omega.S$}$$

and

$$\bigwedge_{\substack{t\in\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E}\}\\ 0<x<y<z<1}} \big[\mathrm{Fr}(T=t, X\approx x, Y\approx y, Z\approx z) = Q_{n+1}(t;x,y,z)\,dx\,dy\,dz\big]. \qquad (\omega.T)$$

The assertions that describe the remaining mass are just the summary assertions with the $k = n+1$ portion stripped off:

$$\bigwedge_{\substack{1\le k\le n\\ s\in\{\mathsf{F},\mathsf{G}\}\\ 0<v<w<1}} \big[\mathrm{Fr}(K=k, S=s, V\approx v, W\approx w) = P_k(s;v,w)\,dv\,dw\big] \qquad (\beta.S)$$

and

$$\bigwedge_{\substack{2\le k\le n\\ t\in\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E}\}\\ 0<x<y<z<1}} \big[\mathrm{Fr}(K=k, T=t, X\approx x, Y\approx y, Z\approx z) = Q_k(t;x,y,z)\,dx\,dy\,dz\big]. \qquad (\beta.T)$$

We really needn't bother remembering the T portion of the β assertion, since the first case-statement is about to reset the values of the three-key variables from the current values of the two-key variables. The only reason that our summary assertion has both S and T portions, in fact, is that we want to be able to support both of the output assertions $(\omega.S)$ and $(\omega.T)$. Therefore, we might as well drop the assertion $(\beta.T)$. This has the advantage that the remaining assertion $(\beta.S)$ cuts the loop, and is differentially disjoint vanilla; hence we can stop worrying about feasibility.

The mass described by $(\beta.S)$ then enters the loop body, where the first action is the random choice of a value for R. This choice affects the atomic assertions by adding the conjunct $R\approx r$ to the predicate and the factor dr to the right hand side. That is, the proper assertion for just after this random assignment is

$$\bigwedge_{\substack{1\le k\le n\\ s\in\{\mathsf{F},\mathsf{G}\}\\ 0<v<w<1\\ 0<r<1}} \big[\mathrm{Fr}(K=k, S=s, V\approx v, W\approx w, R\approx r) = P_k(s;v,w)\,dv\,dw\,dr\big]. \qquad (\gamma.S)$$

The mass described by $(\gamma.S)$ now goes through the case-statement that performs the insertion. Assertion $(\gamma.S)$ splits into six disjoint and exhaustive pieces, depending upon the value of s and the rank of r (to be precise, exhaustive except for a set of measure zero). We shall consider only the first case

$$\bigwedge_{\substack{1\le k\le n\\ 0<r<v<w<1}} \big[\mathrm{Fr}(K=k, V\approx v, W\approx w, R\approx r) = P_k(\mathsf{F};v,w)\,dv\,dw\,dr\big], \qquad (\delta.S)$$

since the others are similar; on this first branch, the relations $S = \mathsf{F}$ and $0 < R < V < W < 1$ are Floyd-Hoare facts. This packet of mass is about to be subjected to the assignments

$$T \leftarrow \mathsf{A};\ X \leftarrow R;\ Y \leftarrow V;\ Z \leftarrow W.$$

To prepare for them, we can change variables in ($\delta.S$) to get the equivalent assertion

$$\bigwedge_{\substack{1 \le k \le n \\ 0<x<y<z<1}} \bigl[\mathrm{Fr}(K = k, V \approx y, W \approx z, R \approx x) = P_k(\mathsf{F}; y, z)\, dy\, dz\, dx\bigr].$$

An axiom of assignment then shows that, after the assignments, the assertion

$$\bigwedge_{\substack{1 \le k \le n \\ 0<x<y<z<1}} \bigl[\mathrm{Fr}(K = k, T = \mathsf{A}, Y \approx y, Z \approx z, X \approx x) = P_k(\mathsf{F}; y, z)\, dy\, dz\, dx\bigr]. \qquad (\epsilon.T)$$

holds. Note that, through the magic of the assignment statement, the S-assertion ($\delta.S$) has supported the T-assertion ($\epsilon.T$). This is natural enough, since the structure of the three-key tree after the insertion is determined by the structure of the two-key tree before.

The other five arms of the case-statement are similar. When these six flows recombine at the end of the insertion case-statement, the six versions of ($\epsilon.T$) will produce an assertion ($\varsigma.T$) that has the same general form as the T-assertion portion ($\sigma.T$) of the summary assertion. In particular, if we determine the function Q_{k+1} from the function P_k by the appropriate relations, the six versions of ($\epsilon.T$) will add up to

$$\bigwedge_{\substack{1 \le k \le n \\ t \in \{\mathsf{A,B,C,D,E}\} \\ 0<x<y<z<1}} \bigl[\mathrm{Fr}(K = k, T = t, X \approx x, Y \approx y, Z \approx z) = Q_{k+1}(t; x, y, z)\, dx\, dy\, dz\bigr]. \qquad (\varsigma.T)$$

In order to make this happen, we need to satisfy for $k \ge 1$ and $0 < x < y < z < 1$ the relations

$$\begin{aligned} Q_{k+1}(\mathsf{A}; x, y, z) &= P_k(\mathsf{F}; y, z) \\ Q_{k+1}(\mathsf{B}; x, y, z) &= P_k(\mathsf{F}; x, z) \\ Q_{k+1}(\mathsf{C}; x, y, z) &= P_k(\mathsf{F}; x, y) + P_k(\mathsf{G}; y, z) \\ Q_{k+1}(\mathsf{D}; x, y, z) &= P_k(\mathsf{G}; x, z) \\ Q_{k+1}(\mathsf{E}; x, y, z) &= P_k(\mathsf{G}; x, y); \end{aligned} \qquad (5.16)$$

these relations are the exact analogs of Jonassen and Knuth's Equation (2.1).

The insertion phase of the loop body is the portion from β to ς that we have just traced. The deletion phase coming up next will take us from ς to κ. Moving our assertions through the deletion phase of the loop body is based on the same principles, but is somewhat more complex. Moving assertion ($\varsigma.T$) through the assignment to L produces the assertion

$$\bigwedge_{\substack{1 \le k \le n \\ t \in \{\mathsf{A,B,C,D,E}\} \\ 0<x<y<z<1 \\ \ell \in \{\mathsf{X,Y,Z}\}}} \bigl[\mathrm{Fr}(K = k, T = t, L = \ell, X \approx x, Y \approx y, Z \approx z) = \tfrac{1}{3} Q_{k+1}(t; x, y, z)\, dx\, dy\, dz\bigr].$$

$$(\eta.T)$$

Assertion $(\eta.T)$ now enters the second **case**-statement, where it is split into fifteen disjoint and exhaustive pieces, depending upon the values of t and ℓ. Again, we shall only go through the first arm of the **case**-statement in detail; the mass entering it is described by

$$\bigwedge_{\substack{1\le k\le n\\ 0<x<y<z<1}} \left[\mathrm{Fr}(K=k, X\approx x, Y\approx y, Z\approx z)=\tfrac{1}{3}Q_{k+1}(\mathsf{A}; x,y,z)\,dx\,dy\,dz\right]. \qquad (\theta.T)$$

On this first arm, the relations $T=\mathsf{A}$ and $L=\mathsf{X}$ are Floyd-Hoare facts. The mass described by $(\theta.T)$ is about to undergo the assignments

$$S\leftarrow \mathsf{F};\ V\leftarrow Y;\ W\leftarrow Z.$$

We prepare for these by rewriting $(\theta.T)$ in the equivalent form

$$\bigwedge_{\substack{1\le k\le n\\ 0<x<v<w<1}} \left[\mathrm{Fr}(K=k, X\approx x, Y\approx v, Z\approx w)=\tfrac{1}{3}Q_{k+1}(\mathsf{A}; x,v,w)\,dx\,dv\,dw\right].$$

This assertion and an appropriate axiom of assignment allow us to conclude that the assertion

$$\bigwedge_{\substack{1\le k\le n\\ 0<x<v<w<1}} \left[\mathrm{Fr}(K=k, S=\mathsf{F}, X\approx x, V\approx v, W\approx w)=\tfrac{1}{3}Q_{k+1}(\mathsf{A}; x,v,w)\,dx\,dv\,dw\right] \qquad (\iota.S')$$

holds after the assignment; in this case, a T-assertion has changed into an S-assertion. Now, the information given by the assertion $(\iota.S')$ is a little bit too detailed, because its atomic assertions specify the joint distributions of K, S, V, W, and X; this is why we called it $(\iota.S')$ instead of $(\iota.S)$. Since the variable X is merely storing the value of the deleted key, we would just as soon throw away that aspect of the joint distribution. We do this, of course, by integrating over all x; the only nonzero values come for x in the range $0<x<v$, and we deduce that the following assertion $(\iota.S)$ also holds at the end of the first arm of the second **case**-statement:

$$\bigwedge_{\substack{1\le k\le n\\ 0<v<w<1}} \left[\mathrm{Fr}(K=k, S=\mathsf{F}, V\approx v, W\approx w)=\tfrac{1}{3}\left(\int_0^v Q_{k+1}(\mathsf{A}; x,v,w)\,dx\right)dv\,dw\right]. \qquad (\iota.S)$$

At the end of the deletion **case**-statement, we would expect the fifteen analogs of $(\iota.S)$ to combine together to support the S portion of the summary assertion. But recall that the summary assertion also has a T portion. We have to have assertions that will preserve our current information about the values of the three-key variables through the rest of the loop body, just to verify that nothing happens to them. In particular, we need an assertion $(\iota.T)$. That is no problem; since the assertion $(\theta.T)$ does not mention the values of the variables that are being reset, a trivial instance of the Assignment Axiom Schema shows us that the assertion

$(\theta.T)$ will remain true at the point ι. Therefore, $(\theta.T)$ deserves the new name $(\iota.T)$:

$$\bigwedge_{\substack{1\le k\le n\\ 0<x<y<z<1}} \big[\mathrm{Fr}(K=k, X\approx x, Y\approx y, Z\approx z) = \tfrac{1}{3}Q_{k+1}(\mathsf{A}; x, y, z)\,dx\,dy\,dz\big]. \qquad (\iota.T)$$

We shall now consider the end of the second **case**-statement, where the fifteen analogs of $(\iota.S)$ and of $(\iota.T)$ combine. The analogs of $(\iota.T)$ just recombine into $(\eta.T)$, since nothing happened to any of the three-key variables during the second **case**-statement. If we then throw away the information about L by summing, we find ourselves all the back at the assertion $(\varsigma.T)$. Thus, the assertion $(\varsigma.T)$ deserves the new name $(\kappa.T)$:

$$\bigwedge_{\substack{1\le k\le n\\ t\in\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E}\}\\ 0<x<y<z<1}} \big[\mathrm{Fr}(K=k, T=t, X\approx x, Y\approx y, Z\approx z) = Q_{k+1}(t; x, y, z)\,dx\,dy\,dz\big]. \qquad (\kappa.T)$$

We went through a lot of work in the process of showing that this assertion passes through the deletion **case**-statement unchanged. If we extended the frequency system with a **case**-statement analog of the Irrelevant Conditional Rule, we might have been able to show in one magnificent step that, since the assignments in the second **case**-statement do not affect the variables described by T-assertions, the assertion $(\varsigma.T)$ will hold at κ if it holds at ς.

The fifteen analogs of $(\iota.S)$ combine into something that looks quite a bit like the S-assertion portion $(\sigma.S)$ of the summary assertion. In fact, if we define the function P_{k+1} from the function Q_{k+1} by the appropriate relations, the analogs of $(\iota.S)$ will combine into

$$\bigwedge_{\substack{1\le k\le n\\ s\in\{\mathsf{F},\mathsf{G}\}\\ 0<v<w<1}} \big[\mathrm{Fr}(K=k, S=s, V\approx v, W\approx w) = P_{k+1}(s; v, w)\,dv\,dw\big]. \qquad (\kappa.S)$$

In order to make this happen, we must arrange that the following relations, which are just Equation (2.2) of Jonassen and Knuth, are satisfied for all $k\ge 1$ and $0<v<w<1$:

$$\begin{aligned}
P_{k+1}(\mathsf{F}; v, w) = {}& \tfrac{1}{3}\int_0^v \big((Q_{k+1}(\mathsf{A}; u, v, w) + Q_{k+1}(\mathsf{B}; u, v, w)\big)\,du\\
&+ \tfrac{1}{3}\int_v^w \big((Q_{k+1}(\mathsf{A}; v, u, w) + Q_{k+1}(\mathsf{B}; v, u, w) + Q_{k+1}(\mathsf{C}; v, u, w)\big)\,du\\
&+ \tfrac{1}{3}\int_w^1 \big((Q_{k+1}(\mathsf{A}; v, w, u) + Q_{k+1}(\mathsf{C}; v, w, u)\big)\,du\\
P_{k+1}(\mathsf{G}; v, w) = {}& \tfrac{1}{3}\int_0^v \big((Q_{k+1}(\mathsf{C}; u, v, w) + Q_{k+1}(\mathsf{D}; u, v, w) + Q_{k+1}(\mathsf{E}; u, v, w)\big)\,du\\
&+ \tfrac{1}{3}\int_v^w \big((Q_{k+1}(\mathsf{D}; v, u, w) + Q_{k+1}(\mathsf{E}; v, u, w)\big)\,du\\
&+ \tfrac{1}{3}\int_w^1 \big((Q_{k+1}(\mathsf{B}; v, w, u) + Q_{k+1}(\mathsf{D}; v, w, u) + Q_{k+1}(\mathsf{E}; v, w, u)\big)\,du.
\end{aligned} \qquad (5.17)$$

The next thing that happens is that the loop control variable K is incremented. Coming out of the body of the loop, we have mass that is described by both $(\kappa.S)$ and $(\kappa.T)$. To prepare for the incrementation of K, we can rewrite these in the equivalent form

$$\bigwedge_{\substack{2\le k\le n+1\\ s\in\{\mathsf{F},\mathsf{G}\}\\ 0<v<w<1}} \bigl[\mathrm{Fr}(K+1=k, S=s, V\approx v, W\approx w) = P_k(s;v,w)\,dv\,dw\bigr]$$

and

$$\bigwedge_{\substack{2\le k\le n+1\\ t\in\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E}\}\\ 0<x<y<z<1}} \bigl[\mathrm{Fr}(K+1=k, T=t, X\approx x, Y\approx y, Z\approx z) = Q_k(t;x,y,z)\,dx\,dy\,dz\bigr].$$

An assignment axiom then shows that, after K is incremented, we may conclude

$$\bigwedge_{\substack{2\le k\le n+1\\ s\in\{\mathsf{F},\mathsf{G}\}\\ 0<v<w<1}} \bigl[\mathrm{Fr}(K=k, S=s, V\approx v, W\approx w) = P_k(s;v,w)\,dv\,dw\bigr]$$

and

$$\bigwedge_{\substack{2\le k\le n+1\\ t\in\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E}\}\\ 0<x<y<z<1}} \bigl[\mathrm{Fr}(K=k, T=t, X\approx x, Y\approx y, Z\approx z) = Q_k(t;x,y,z)\,dx\,dy\,dz\bigr].$$

Our final task is to add in the mass that is entering the loop for the first time, and to check that the resulting assertions match the summary assertions $(\sigma.S)$ and $(\sigma.T)$ with which we began. Comparing what we currently have with what we want, we find that the T portion of what we have exactly matches the assertion $(\sigma.T)$, while the S portion matches all of $(\sigma.S)$ except for the atomic assertions with $k=1$. Since the new input mass all has $K=1$, we shall be done as long as that input mass exactly accounts for the $k=1$ conjuncts in $(\sigma.S)$. We are assuming that the program InsertDelete is entered with the variables S, V, and W describing the tree that results from two successive insertions of a random key into an initially empty tree. Therefore, we need only add to our constraints on the functions P_k and Q_k the initial condition

$$P_1(\mathsf{F};v,w) = P_1(\mathsf{G};v,w) = 1 \tag{5.18}$$

for $0<v<w<1$; this is Equation (2.4) of Jonassen and Knuth.

Working in the frequency system, we have shown that the functions P_k and Q_k that satisfy Equations (5.16), (5.17) and (5.18) describe the joint distribution of the tree shapes and keys in the program InsertDelete. This completes the dynamic phase of the data analysis of InsertDelete. The static phase is a considerably greater challenge, since the recurrences are rather formidable. The reader is referred to the paper by Jonassen and Knuth for the interesting tale of their solution [16].

Chapter 6. Beyond the Frequency System

Restricted and Arbitrary Goto's.

In our development of the frequency system, we limited ourselves to single exit loops. To what extent can we allow more general control structures? After all, Floyd-Hoare systems are able to handle arbitrary **goto**'s and recursive procedures. To investigate what it would take to extend the frequency system to include these constructs, let us start with a simple one, the **exit**-statement. If a statement S is labelled with the label L, then a statement of the form "**exit** L" occurring within S causes control to jump to the end of S. The rule that handles **exit**-statements in a Floyd-Hoare system is the following:

$$\frac{\{Q\}\ \textbf{exit}\ L\ \{\textsc{false}\} \vdash \{P\}\, S\, \{Q\}}{\vdash \{P\}\ L\colon S\ \{Q\}}.$$

The formula occurring before the "$\vdash$" in the premise may be used as an axiom during the derivation of the formula after that "$\vdash$". This temporary axiom allows us to deduce inside of the statement S that any control that enters an **exit**-statement will never emerge. In order to apply the temporary axiom, however, we must guarantee that any control that executes an **exit**-statement will be in a state that satisfies the predicate Q, so that the postassertion of the labelled statement will not be violated.

If we consider what the flowchart of a program using **exit**-statements looks like, we can see that putting an Exit Rule into the frequency system is somewhat more complex. In particular, at the end of the labelled statement is a join in the corresponding plumbing network, where different flows converge. There is the flow that is exiting the statement S in the normal way, and there are also all of the flows that have jumped to this join from **exit**-statements throughout the body of S. In the Floyd-Hoare world, the logical connective that implements a join is "or", and that is reflected in the Exit Rule above. But in the frequency system, the connective for joins is addition, an arithmetic connective. Therefore, the appropriate postassertion for the labelled statement is the sum of the normal postassertion for the body S and all of the preconditions of **exit**-statements within S. Unfortunately, it is not easy to see how to write a rule that incorporates this insight. The formulas of Floyd-Hoare systems, and of the frequency system to date, involve a precondition, a program with a single entry and single exit, and a postassertion. In order to describe the correct postassertion of the labelled statement, we must be able to refer to all of the preconditions of the **exit**-statements inside S.

One possible solution for this problem is to allow our system to deal with a more general kind of augmented program, in particular, a program with associated assertions not only at the beginning and end, but also interspersed throughout. The intermediate assertions would record the assertions that we employed in analyzing the smaller pieces of the current program. In this extension of the frequency system, it would be straightforward to write down an Exit Rule. In fact, even arbitrary **goto**'s would not present too many difficulties. The postassertion of each **goto**-statement itself would be $\mathrm{Fr}(\textsc{true}) = 0$, the analog of the Floyd-Hoare predicate

FALSE, while the postassertion of each label would be the sum of the precondition for that label and the preconditions of all of the **goto**'s that jump to that label. The correct techniques are straightforward if one thinks about programs in terms of their flowcharts.

Of course, arbitrary **goto**'s can implement loops, and so we must carry over to this new environment the techniques that we have developed for retaining the soundness of the system. First, remember that fictitious mass is an ever-present possibility. Therefore, even though one is dealing with augmented programs that have assertions sprinkled throughout, there is no guarantee that any assertions other than those outside of all loops actually describe the corresponding demon reports. The other assertions will describe the true behavior, but may also describe some fictitious mass. In order to prevent time bombs, we must insist that every loop in the flowchart be cut by at least one feasible assertion, that is, one assertion whose characteristic subset of $\mathfrak{F}^+$ is non-empty. And finally, the postassertion of the entire program must be closed. If we make these restrictions, a straightforward generalization of the **while**-loop theorem of Chapter 4 will demonstrate that the input-output frequentistic behavior of the entire program is correctly described by its precondition and postassertion. Since the loop breaking assertions are feasible, we shall be able to trace any finite path through the flowchart. By tracing longer and longer paths, we can guarantee that the output assertion covers more and more of what really happens. Then closure will allow us to take the necessary limit.

InsertionSort.

We shall now consider the algorithm that performs a straight insertion sort. We have postponed this example until now because most codings of a straight insertion sort involve either an **exit** or a **goto**. This example is particularly interesting, because Ben Wegbreit presented one version of InsertionSort as the primary example of the use of his system. After we do a performance analysis of InsertionSort in the frequency system, we shall have a good opportunity to compare the two systems in action.

The following program, which we shall call InsertionSort, implements the algorithm of the same name [20]. The Jth element of the input array X of length $n \geq 1$ is compared with $(J-1)$st, $(J-2)$nd, and so on, until its proper final position is found:

```
for $J$ from 2 to $n$ do
               $I \leftarrow J-1$; $Y \leftarrow X[J]$;
        nextI: if $Y \geq X[I]$ then goto nextJ fi;
               $X[I+1] \leftarrow X[I]$; $I \leftarrow I-1$;
               if $I > 0$ then goto nextI fi;
        nextJ: $X[I+1] \leftarrow Y$;
od.
```

As in the InsertDelete example, we should really have a program variable N as the upper limit of the **for**-loop, and then start the program in a state in which the Floyd-Hoare fact $N = n$ holds; but we won't.

The program InsertionSort has two performance parameters of interest. Adopting Knuth's notations [20], they are the number A of times that the if-test $I > 0$ comes out FALSE, and the number B of times that the assignment $X[I+1] \leftarrow X[I]$ is performed. Our first step is to add two counter variables, called A and B respectively, that will keep track of these quantities. The resulting monitored program with appropriate control points labelled is:

```
        $A \leftarrow 0$; $B \leftarrow 0$;
        $[\![\alpha]\!]$ for $J$ from 2 to $n$ $[\![\sigma]\!]$ do
                    $[\![\beta]\!]$ $I \leftarrow J-1$; $Y \leftarrow X[J]$; $[\![\gamma]\!]$
            nextI:  $[\![\delta]\!]$ if $Y \geq X[I]$ then $[\![\epsilon]\!]$ goto nextJ fi;
                    $[\![\varsigma]\!]$ $X[I+1] \leftarrow X[I]$; $B \leftarrow B+1$; $I \leftarrow I-1$;
                    $[\![\eta]\!]$ if $I > 0$ then $[\![\theta]\!]$ goto nextI fi;
                    $[\![\iota]\!]$ $A \leftarrow A+1$; $[\![\kappa]\!]$
            nextJ:  $[\![\lambda]\!]$ $X[I+1] \leftarrow Y$; $[\![\mu]\!]$
          od $[\![\omega]\!]$.
```

Once again, we have named certain control points with Greek letters in alphabetic order, except that σ, α, and ω are reserved for the summary, input, and output assertions of the **for**-loop respectively.

The input to InsertionSort is a random permutation, and we shall use the continuous model where the input array consists of n independent random variables chosen from the uniform distribution U on $[0,1)$. Our assertions in this analysis will be more complex than those of the InsertDelete and FindMax analyses because we shall have to keep track of the joint distributions of a non-bounded number of variables—in particular, of all of the array elements. In our analyses of InsertDelete and FindMax, we were able to choose new values and integrate out old ones incrementally, so that we only had to deal with a few values at a time. But for InsertionSort, we must keep track of everything at once.

In our current model, a random input permutation corresponds to a random point in the n-dimensional unit hypercube. Any correct sorting program should take us from the input state

$$\bigwedge_{\langle x_1,x_2,\ldots,x_n\rangle\in(0,1)^n} \left[\operatorname{Fr}(X[1]\approx x_1, X[2]\approx x_2,\ldots,X[n]\approx x_n) = dx_1\,dx_2\ldots dx_n\right] \tag{6.1}$$

to the output state

$$\bigwedge_{0<x_1<x_2<\cdots<x_n<1} \left[\operatorname{Fr}(X[1]\approx x_1, X[2]\approx x_2,\ldots,X[n]\approx x_n) = n!\,dx_1\,dx_2\ldots dx_n\right]. \tag{6.2}$$

The job of the sorting program is to fold space so that all $n!$ sub-regions of the hypercube in which the ordering relationships among the coodinates are constant end up superimposed. We can ignore the possibility of equal keys throughout, since this event happens only with probability zero.

Assertions (6.1) and (6.2) would be the precondition and postassertion of a data analysis of InsertionSort. Since we are doing a performance analysis, our postassertion will be more complex. In order to save space when writing assertions, we shall adopt an array slicing notation; the expression $X[i{:}j]$ stands for the portion of the X array from the ith through the jth elements inclusive, and $x_{[i:j]}$ is the analogous expression in the subscript form. With this notation, the predicate in assertions (6.1) and (6.2) could be written $X[1{:}n] \approx x_{[1:n]}$. We shall also make the convention that an inequality applied to an array slice applies to all of its elements; thus, the index restrictions in assertion (6.1) could be written $0 < x_{[1:n]} < 1$ instead of $\langle x_1, x_2, \ldots, x_n\rangle \in (0,1)^n$.

As in our treatment of InsertDelete, we shall name the assertions by the control points at which they apply. This analysis will be presented in somewhat larger steps than the preceding ones; the appropriate assertions will be accompanied by only a few comments. The input assertion at α is

$$\bigwedge_{0<x_{[1:n]}<1} \big[\mathrm{Fr}(X[1{:}n] \approx x_{[1:n]}, A = 0, B = 0) = dx_1\,dx_2 \ldots dx_n\big]. \qquad (\alpha)$$

With our growing experience in using the frequency system, it doesn't take much thought to decide upon an appropriate summary assertion for the **for**-loop. The summary assertion should describe the joint distributions of the array elements and the variables J, A, and B. The Floyd-Hoare property that describes the array at the point σ is that the first $J-1$ positions are in sorted order. With these things in mind, we can decide upon the summary assertion

$$\bigwedge_{\substack{2\le j\le n+1\\ 0<x_1<\cdots<x_{j-1}<1\\ 0<x_{[j:n]}<1\\ a\ge 0,\quad b\ge 0}} \big[\mathrm{Fr}(X[1{:}n] \approx x_{[1:n]}, J = j, A = a, B = b) = p_{a,b,j}\,dx_1\,dx_2 \ldots dx_n\big] \qquad (\sigma)$$

where the coefficients $p_{a,b,j}$ are new unknowns.

In order for the input assertion (α) to support the $j = 2$ portion of the summary assertion (σ), we must arrange that the coefficients $p_{a,b,j}$ satisfy the initial condition

$$p_{a,b,2} = \delta_{a0}\delta_{b0}. \qquad (6.3)$$

The mass described by the $j = n+1$ portion of the summary assertion exits the loop, and forms the output assertion (ω):

$$\bigwedge_{\substack{0<x_1<\cdots<x_n<1\\ a\ge 0,\quad b\ge 0}} \big[\mathrm{Fr}(X[1{:}n] \approx x_{[1:n]}, A = a, B = b) = p_{a,b,n+1}\,dx_1\,dx_2 \ldots dx_n\big]. \qquad (\omega)$$

The rest of the summary assertion mass enters the loop to support the assertion (β) at the beginning of the loop body:

$$\bigwedge_{\substack{2\le j\le n\\ 0<x_1<\cdots<x_{j-1}<1\\ 0<x_{[j:n]}<1\\ a\ge 0,\ b\ge 0}} \big[\mathrm{Fr}(X[1{:}n] \approx x_{[1:n]}, J = j, A = a, B = b) = p_{a,b,j}\,dx_1\,dx_2 \ldots dx_n\big]. \qquad (\beta)$$

The assignments to I and Y then put us in the state

$$\bigwedge_{\substack{2\le j\le n,\quad i=j-1\\ 0<x_1<\cdots<x_i<1\\ 0<x_{[j+1:n]}<1\\ a\ge 0,\quad b\ge 0,\quad 0<y<1}} \left[\mathrm{Fr}\begin{pmatrix} X[1{:}i]\approx x_{[1:i]}, X[j+1{:}n]\approx x_{[j+1:n]},\\ Y\approx y, I=i, J=j, A=a, B=b\end{pmatrix} = p_{a,b,j}\,dx_1\ldots dx_i\,dy\,dx_{j+1}\ldots dx_n\right]. \qquad (\gamma)$$

The flow that (γ) describes has to do its share in supporting the assertion at the point δ. Note that the assertion at δ cuts a loop generated by a **goto**-statement, the inner loop in which I varies. If we wanted to proceed in the most straightforward way, we would now invent a summary assertion for this inner loop that involved a four-parameter family of coefficients $q_{a,b,i,j}$. We shall save some writing, however, by realizing that the variables I and B are manipulated in a very simple way in this inner loop. Hence, we can avoid going to four-parameter coefficients if we are smart enough to invent the following asssertion at the point δ:

$$\bigwedge_{\substack{2\le j\le n,\quad 0<i<j\\ 0<x_1<\cdots<x_i<x_{i+2}\\ 0<y<x_{i+2}<\cdots<x_j<1\\ 0<x_{[j+1:n]}<1\\ a\ge 0,\quad b\ge 0}} \left[\mathrm{Fr}\begin{pmatrix} X[1{:}i]\approx x_{[1:i]}, X[i+2{:}n]\approx x_{[i+2:n]}\\ Y\approx y, I=i, J=j, A=a, B=b\end{pmatrix} = p_{a,b-j+i+1,j}\,dx_1\ldots dx_i\,dy\,dx_{i+2}\ldots dx_n\right]. \qquad (\delta)$$

To avoid a special case, we take the expression x_{n+1} to mean 1. Note that (γ) supports all of the $i=j-1$ mass in (δ). The assertion (δ) cuts both of the program loops, and is differentially disjoint vanilla; hence, we have satisfied the feasibility restriction.

Of the mass described by (δ), the portion where $Y\ge X[I]$ moves on to the point ϵ:

$$\bigwedge_{\substack{2\le j\le n,\quad 0<i<j\\ 0<x_1<\cdots<x_i<y\\ y<x_{i+2}<\cdots<x_j<1\\ 0<x_{[j+1:n]}<1\\ a\ge 0,\quad b\ge 0}} \left[\mathrm{Fr}\begin{pmatrix} X[1{:}i]\approx x_{[1:i]}, X[i+2{:}n]\approx x_{[i+2:n]},\\ Y\approx y, I=i, J=j, A=a, B=b\end{pmatrix} = p_{a,b-j+i+1,j}\,dx_1\ldots dx_i\,dy\,dx_{i+2}\ldots dx_n\right]. \qquad (\epsilon)$$

The rest of (δ) then moves on to the point ς:

$$\bigwedge_{\substack{2\le j\le n,\quad 0<i<j\\ 0<x_1<\cdots<x_i\\ 0<y<x_i<x_{i+2}<\cdots<x_j<1\\ 0<x_{[j+1:n]}<1\\ a\ge 0,\quad b\ge 0}} \left[\mathrm{Fr}\begin{pmatrix} X[1{:}i]\approx x_{[1:i]}, X[i+2{:}n]\approx x_{[i+2:n]}\\ Y\approx y, I=i, J=j, A=a, B=b\end{pmatrix} = p_{a,b-j+i+1,j}\,dx_1\ldots dx_i\,dy\,dx_{i+2}\ldots dx_n\right]. \qquad (\varsigma)$$

Moving this assertion through the next three assignments is tedious, but not tricky; the result at control point η is:

$$\bigwedge_{\substack{2\le j\le n,\quad 0\le i<j-1\\ 0<x_1<\cdots<x_i<x_{i+2}\\ 0<y<x_{i+2}<\cdots<x_j<1\\ 0<x_{[j+1:n]}<1\\ a\ge 0,\quad b\ge 1}} \left[\mathrm{Fr}\begin{pmatrix} X[1{:}i]\approx x_{[1:i]}, X[i+2{:}n]\approx x_{[i+2:n]}\\ Y\approx y, I=i, J=j, A=a, B=b\end{pmatrix} = p_{a,b-j+i+1,j}\,dx_1\ldots dx_i\,dy\,dx_{i+2}\ldots dx_n\right]. \qquad (\eta)$$

The $i > 0$ portion of this mass is described by (θ):

$$\bigwedge_{\substack{2\le j\le n,\quad 0<i<j-1\\ 0<x_1<\cdots<x_i<x_{i+2}\\ 0<y<x_{i+2}<\cdots<x_j<1\\ 0<x_{[j+1:n]}<1\\ a\ge 0,\quad b\ge 1}} \left[\operatorname{Fr}\begin{pmatrix} X[1{:}i]\approx x_{[1:i]}, X[i+2{:}n]\approx x_{[i+2:n]}\\ Y\approx y, I=i, J=j, A=a, B=b\end{pmatrix} = p_{a,b-j+i+1,j}\,dx_1\ldots dx_i\,dy\,dx_{i+2}\ldots dx_n\right]. \tag{θ}$$

Comparing (θ) with (δ), we can see that (θ) is ready to support almost all of the $i \neq j-1$ portion of the (δ) mass; recall that (γ) already supports the $i = j-1$ portion. The only difficulty is that the index b has the lower limit 1 in (θ), while it has the lower limit 0 in (δ). We can make sure that this does not cause a difficulty by merely agreeing to demand that the coefficients $p_{a,b,j}$ satisfy the boundary condition

$$p_{a,b,j} = 0 \qquad \text{for} \quad b < 0. \tag{6.4}$$

The $i = 0$ portion of the (η) mass falls through to form the assertion (ι):

$$\bigwedge_{\substack{2\le j\le n\\ 0<y<x_2<\cdots<x_j<1\\ 0<x_{[j+1:n]}<1\\ a\ge 0,\quad b\ge 0}} \left[\operatorname{Fr}\begin{pmatrix} Y\approx y, X[2{:}n]\approx x_{[2:n]}\\ I=0, J=j, A=a, B=b\end{pmatrix} = p_{a,b-j+1,j}\,dy\,dx_2\ldots dx_n\right]. \tag{ι}$$

We have replaced the lower limit on b by 0 in this assertion as well, because of Equation (6.4). Next, the assertion (ι) passes through the assignment $A \leftarrow A+1$ to become (κ):

$$\bigwedge_{\substack{2\le j\le n\\ 0<y<x_2<\cdots<x_j<1\\ 0<x_{[j+1:n]}<1\\ a\ge 1,\quad b\ge 0}} \left[\operatorname{Fr}\begin{pmatrix} Y\approx y, X[2{:}n]\approx x_{[2:n]}\\ I=0, J=j, A=a, B=b\end{pmatrix} = p_{a-1,b-j+1,j}\,dy\,dx_2\ldots dx_n\right]. \tag{κ}$$

The assertions (κ) and (ϵ) should add together to produce the assertion (λ). Since (κ) almost exactly fills in the $i = 0$ portion of (ϵ), we shall attempt to adjust things so that that fit is exact. The problems center around the index variable a. In (κ), we conjoin over $a \geq 1$, and the first index of the p coefficient is $a-1$; in (ϵ), however, we conjoin over $a \geq 0$, and that first index is simply a. We can solve these problems if we both demand the boundary condition

$$p_{a,b,j} = 0 \qquad \text{for} \quad a < 0, \tag{6.5}$$

and define the assertion (λ) to be

$$\bigwedge_{\substack{2\le j\le n,\quad 0\le i<j\\ 0<x_1<\cdots<x_i<y\\ y<x_{i+2}<\cdots<x_j<1\\ 0<x_{[j+1:n]}<1\\ a\ge 0,\quad b\ge 0}} \left[\operatorname{Fr}\begin{pmatrix} X[1{:}i]\approx x_{[1:i]}, X[i+2{:}n]\approx x_{[i+2:n]},\\ Y\approx y, I=i, J=j, A=a, B=b\end{pmatrix} = p_{a-\delta_{i0},b-j+i+1,j}\,dx_1\ldots dx_i\,dy\,dx_{i+2}\ldots dx_n\right]. \tag{λ}$$

The assignment $X[I+1] \leftarrow Y$ allows us to clean up a little bit, as we move from the assertion (λ) to assertion (μ):

$$\bigwedge_{\substack{2\le j\le n,\ \ 0\le i<j\\ 0<x_1<\cdots<x_j<1\\ 0<x_{[j+1:n]}<1\\ a\ge 0,\ \ b\ge 0}} \left[\mathrm{Fr}\begin{pmatrix} X[1{:}n]\approx x_{[1:n]}, \\ I=i, J=j, A=a, B=b\end{pmatrix} = p_{a-\delta_{i0},b-j+i+1,j}\,dx_1\ldots dx_n\right]. \tag{μ}$$

All that is left is for (μ) to go through the implicit assignment $J \leftarrow J+1$ that increments the loop index, and then do its part in supporting the summary assertion. Since the summary assertion does not contain any information about the distribution of I, we shall first sum the assertion (μ) on i, getting

$$\bigwedge_{\substack{2\le j\le n\\ 0<x_1<\cdots<x_j<1\\ 0<x_{[j+1:n]}<1\\ a\ge 0,\ \ b\ge 0}} \left[\mathrm{Fr}\begin{pmatrix} X[1{:}n]\approx x_{[1:n]}, \\ J=j, A=a, B=b\end{pmatrix} = \left(\sum_{0\le i<j} p_{a-\delta_{i0},b-j+i+1,j}\right) dx_1\ldots dx_n\right].$$

The incrementation of J just changes this into

$$\bigwedge_{\substack{3\le j\le n+1\\ 0<x_1<\cdots<x_{j-1}<1\\ 0<x_{[j:n]}<1\\ a\ge 0,\ \ b\ge 0}} \left[\mathrm{Fr}\begin{pmatrix} X[1{:}n]\approx x_{[1:n]}, \\ J=j, A=a, B=b\end{pmatrix} = \left(\sum_{0\le i<j-1} p_{a-\delta_{i0},b-j+i+2,j-1}\right) dx_1\ldots dx_n\right].$$

Recall that the input mass described by (α) is already supporting the $j=2$ portion of the summary assertion (σ). Comparing our current assertion with (σ), we see that our current assertion will precisely support the rest if the appropriate identity holds among the coefficients $p_{a,b,j}$. In particular, we need to demand that

$$p_{a,b,j} = \sum_{0\le i<j-1} p_{a-\delta_{i0},b-j+i+2,j-1};$$

we can rephrase this more conveniently by stating that, for all $n\ge 1$, $a\ge 0$, and $b\ge 0$, we demand

$$p_{a,b,n+1} = p_{a-1,b-n+1,n} + \sum_{2\le i\le n} p_{a,b-n+i,n}. \tag{6.6}$$

This completes the formal performance analysis of Insertion Sort. We have shown that, if the InsertionSort program is started in a state described by (α), then the mass that exits the program will constitute a state described by the postassertion (ω), where the coefficients $p_{a,b,n+1}$

in (ω) are the solutions to the recurrence and side conditions of Equations (6.3) through (6.6). Without a little bit of study of this recurrence, however, it is not immediately clear even how likely InsertionSort is to halt, much less what its average case performance might be. Rather than studying the recurrence formally, we shall instead attempt to put some intuition behind the symbol manipulations that we have performed by interpreting the coefficients $p_{a,b,n+1}$ in more familiar combinatorial terms.

The quantity $p_{a,b,n+1}$ for $n \geq 1$ counts the number of permutations of n distinct numbers that have precisely a left-to-right minima and precisely b inversions; the leftmost element of the permutation is not counted as a left-to-right minimum. This combinatorial interpretation is easily seen to agree with the side conditions (6.3) through (6.5) on the $p_{a,b,j}$ coefficients. The recurrence (6.6) can be explained as follows: break up the permutations on n numbers with a left-to-right minima and b inversions into classes depending upon the position where the largest number occurs, and consider the permutation that remains when that largest element is deleted. Note that the largest number will contribute a left-to-right minimum only if it is the leftmost element; and note that the largest element will be inverted with repect to every element to its right. If the largest number occurs as the leftmost element, the remaining $n-1$ numbers will form a permutation with $a-1$ left-to-right minima, and with $b-(n-1)$ inversions. On the other hand, if the largest element occurs in the ith position counting from the left for $2 \leq i \leq n$, then the remaining numbers will form a permutation with a left-to-right minima and $b-(n-i)$ inversions. This method of counting demonstrates that the quantities $p_{a,b,n+1}$ that we have defined combinatorially do indeed satisfy recurrence (6.6).

The performance of the program InsertionSort depends upon both the number of left-to-right minima A, and the number of inversions B. Thus, recurrence (6.6) might be justly titled *the performance recurrence* for the InsertionSort program. We can derive from (6.6) a separate recurrence for either A or B by summing out the other index. If we use an asterisk in the index positions that are being summed, we find that summing out a leaves us with the recurrence

$$p_{*,b,n+1} = \sum_{1 \leq i \leq n} p_{*,b-n+i,n},$$

which is the recurrence that counts inversions; summing out b gives us the recurrence

$$p_{a,*,n+1} = p_{a-1,*,n} + (n-1)p_{a,*,n},$$

which is just a rescaled version of our old friend, the recurrence for the number of left-to-right maxima (or minima). Finally, if we sum out both a and b, we are left with the recurrence

$$p_{*,*,n+1} = np_{*,*,n},$$

which has the solution $p_{*,*,n+1} = n!$. That is great news for us! It shows that, if we take the output assertion (ω),

$$\bigwedge_{\substack{0<x_1<\cdots<x_n<1 \\ a \geq 0, \quad b \geq 0}} \left[\mathrm{Fr}(X[1{:}n] \approx x_{[1:n]}, A = a, B = b) = p_{a,b,n+1}\, dx_1\, dx_2 \ldots dx_n\right], \qquad (\omega)$$

and integrate out all of the distribution information—the array elements as well as A and B—we shall be left with the assertion $\mathrm{Fr}(\text{TRUE}) = 1$. Hence, the program InsertionSort halts (with probability one, at least). We shall cease our investigation of InsertionSort with this result. If we wanted to know more about the performance of InsertionSort, we would only have to study its performance recurrence (6.6); but that would take us too far afield.

Comparative Systems.

Now that we have seen a performance analysis of the program InsertionSort in the frequency system, we should pause for a moment to compare this with the analysis of the same program in Wegbreit's system. One major difference is the method used for deriving the performance information. We first added counter variables to the program, and then discussed the joint distributions of those counter variables and the program's data. In Wegbreit's analysis, the formal system only discussed the distribution of the data. This data distribution information was used to compute the branching probabilities, from which the performance results were then deduced. It is interesting to note that Wegbreit recommends the use of counter variables for performing formal worst case analyses. Perhaps he was unable to use counter variables in the probabilistic world because his system, based upon probabilities rather than frequencies, had no cure for the Leapfrog problem.

Another major difference concerns the input assumption. We characterized a random permutation as a random point in the n-dimensional hypercube. Wegbreit characterized his input by assuming essentially that its inversion table was random. In particular, neither system seems able to handle the input assumption that analysts usually use, that of a discrete random permutation. Wegbreit's characterization of a random permutation seems somewhat less perspicuous than our own. In addition, since inversions are an important concept in the analysis of InsertionSort, it would be better if possible not to build that concept into the input assumption.

Finally, recall that Wegbreit's system demanded a division of the program variables into a random class and a non-random class. In the particular case of InsertionSort, this division could be made very naturally: the array elements were considered random, while the pointers I and J into that array were treated as non-random. For other programs, such a division might be harder to construct. The purpose of this division is to partition the universe of all of the pellets passing a demon into chunks over which to compute probabilities. Since the frequency system deals in frequencies instead, all of the program variables played the same formal role in our analysis, even though we might have been thinking of them differently.

Procedures.

The frequency system seems to handle the dynamic phase of the analysis of InsertionSort rather neatly. Now that we can handle **exit**'s and **goto**'s, we should devote some energy to thinking about how procedures, recursive and otherwise, might be dealt with in an extension of the frequency system. Non-recursive procedures are not difficult, but recursive ones are another story.

Think about a non-recursive procedure first, say one whose body involves no other procedure calls. Such a procedure could be expanded in line, as if it were a macro. The general frequency system idea of describing everything that ever happens suggests one way to monitor the flow associated with that procedure: at each spot in the procedure body, we describe all of the mass that flows through that spot on all calls to the procedure. Call this the *all-calls technique.*

Let us consider for a moment adopting the all-calls technique. At each statement that calls a procedure, the mass coming into the call statement should be viewed as flowing into the body of the procedure, after appropriate renamings have been performed. We won't worry about the naming issues associated with procedures—call by value versus call by reference and so on—since these issues have been dealt with by the designers of program verification systems, and the issues in an extension of the frequency system would be the same. If we ignore the naming problems, then the precondition that we put on a procedure body should be the sum of all of the flows that enter the statements that call that procedure. This is essentially the same kind of summing operation that went on for **goto**-statements.

But what happens at the end of the procedure body? At that point, all of the mass that has made it through the body must be split up again into the pieces that will constitute the postassertions of the calls on the procedure. Furthermore, it is rather important that the correspondence between where the mass came from and where it goes back to be preserved. Unfortunately, the current frequency system has no real mechanism for keeping track of which pellets going through the procedure body came from which places. One possible solution to this problem would be to replicate the procedure body, and generate one copy for each call of the procedure. Then, we can use each copy to trace the procedure's execution on the mass from only one call, and that will prevent confusion. But this just corresponds to expanding the procedure in line at each place where it is called; if we are willing to do that, of course procedures are no problem. In fact, they aren't procedures at all, they really are macros.

A second possible solution is to make the call stack of the process an explicit part of the process state, one that can be talked about in our assertions. Then, we can make the assertions everywhere in the program discuss the joint distribution of the program data and the state of the stack. This will work, but it corresponds to explicitly implementing procedure call and return by means of a stack, which is also not a pleasant possibility.

We can find a much better solution to the dilemma of keeping track of "who should return to where" if we get back to basics, and think a little about why a piece of code was encapsulated into a procedure in the first place. The object of a procedure is to implement a certain abstract behavior; when we call the procedure, we don't want to worry about how that behavior is achieved, but only about what affect it has on the current state. In the world of the frequency system, the formal meaning of "a behavior" is a linear map from $\mathfrak{F}$ to $\mathfrak{F}$. A procedure should be thought of as an encapsulated piece of code that performs an abstract task; that is, a procedure is a linear map. When we call the procedure, the only things that should be relevant are the properties of this map, not the actual code of the procedure.

This suggests the following scheme: in addition to putting assertions at various places in and around the body of a procedure, we also characterize the effect of the procedure as a whole by means of a pair of assertions describing the corresponding inputs and outputs. The natural place to put these assertions is at the procedure heading. Consider, for example, the procedure Swap, which interchanges its two integer arguments:

```
procedure Swap($I, J$);
  begin
    declare $T$: integer;
    $T \leftarrow I$; $I \leftarrow J$; $J \leftarrow T$;
  end.
```

The abstract frequentistic behavior of the Swap procedure is described by the pair of assertions

$$\big[\mathrm{Fr}(I = i, J = j) = 1\big] \wedge \big[\mathrm{Fr}([I \neq i] \vee [J \neq j]) = 0\big]$$

on input and

$$\big[\mathrm{Fr}(J = i, I = j) = 1\big] \wedge \big[\mathrm{Fr}([J \neq i] \vee [I \neq j]) = 0\big]$$

on output. For some other piece of code about to call the Swap procedure, this pair of assertions has the following meaning: if your current state matches the input assertion for some values of i and j, then your state immediately after the call to Swap will match the output assertion with the same values of i and j. In fact, if your current state is a linear combination of frequentistic states which match the input assertion, then your state after the call to Swap will be the corresponding linear combination of the output assertions; this follows because the semantic meaning of every program is a linear function.

The input and output assertions of a procedure, then, should describe the behavior of that procedure in a typical case. When that procedure is called, the precondition and postassertion of the call should be linear combinations of appropriate instances of the input and output assertions respectively. Once again, we are ignoring all the issues associated with argument passing and renaming. But what assertions should appear in and around the body of the procedure? Since we don't want the callers to know about the body of the procedure, the natural choice is to have the assertions in the procedure body merely discuss the typical execution of the procedure described by its input and output assertions.

These ideas provide a satisfactory solution to the problem of handling non-recursive procedures in an extension of the frequency system. Each procedure is described by an pair of assertions describing its input and output in a general case. A call on the procedure only examines these input and output assertions, and specializes them by taking an appropriate linear combination of instances. Inside the body of a procedure, the usual techniques of the frequency system are used to show that the procedure's input and output assertions correctly describe the effect of the execution of the procedure body in the general case.

But this scheme is not sound for recursive procedures. Consider, for example, the procedure CallMe:

```
procedure CallMe;
  begin
    call CallMe;
  end.
```

The above techniques would allow us to associate with CallMe an arbitrary pair of input-output assertions. Suppose that we claim that the input-output behavior of CallMe is described by the pair of assertions $\langle A, B \rangle$; then, we will be able to verify that the body of CallMe achieves this functional performance by invoking our claim as an assumption. Of course, direct recursions like this don't terminate.

The same phenomenon arises in a Floyd-Hoare analysis of CallMe; but since Floyd-Hoare systems only deal with partial correctness, this phenomenon is tolerable in the Floyd-Hoare world. In the frequency system, we are dealing with strong performance, and hence we cannot allow this sort of thing. The input assertion $\mathrm{Fr}(\text{TRUE}) = 1$ and output assertion $\mathrm{Fr}(\text{TRUE}) = 2$, for example, do not correctly describe the procedure CallMe (or any other program either, for that matter).

In order to handle recursive procedures in an extension of the frequency system, we would have to develop a method of tracing at least some mass as it goes through a complete recursion. By guaranteeing that the assertions that describe this mass are feasible, we would be able to control the loops that arise from recursive programs with the same techniques that have tamed while-loops. But this is easier said than done. As soon as we allow our assertions to describe more than one execution of the procedure, say the top-level execution and one of the recursive calls, we run right back into the difficulty that we can't keep track of which mass is which.

As an example of the bad things that can happen when different mass flows get confused, consider the procedure DoNothing whose body is the empty statement:

```
procedure DoNothing;
  begin nothing end.
```

Suppose that we associate with the procedure DoNothing the input assertion

$$[\mathrm{Fr}(K = k) = 1] \wedge [\mathrm{Fr}(K \neq k) = 0]$$

and the corresponding output assertion

$$[\mathrm{Fr}(K = 1 - k) = 1] \wedge [\mathrm{Fr}(K \neq 1 - k) = 0].$$

According to this pair of assertions, the DoNothing procedure actually achieves the same effect as the assignment statement $K \leftarrow 1 - K$, which is of course nonsense. But, let us suppose that we are working in an extension of the frequency system in which, say, the all-calls technique is employed. And also suppose that there are two consecutive calls on the procedure DoNothing:

on entry to the first call is one gram of mass with $K = 0$, while on entry to the second is one gram of mass with $K = 1$. Coming out of the first call will be one gram with $K = 1$, and coming out of the second will be one gram with $K = 0$, in accordance with the claimed abstract behavior of DoNothing. The problem is that the empty body of DoNothing also happens to look correct. In particular, since the all-calls technique involves simply adding up the descriptions of all of the calls on the procdure, that empty body will have

$$[\mathrm{Fr}(K = 0) = 1] \wedge [\mathrm{Fr}(K = 1) = 1] \wedge [\mathrm{Fr}(K \neq 0, K \neq 1) = 0]$$

as both its precondition and postassertion. Our inability to keep track of which mass came from where causes a faulty collection of assertions to look everywhere locally correct.

In summary, it is not easy to see how to combine the encapsulation that is the essence of a procedure with the global, "describe everything that ever happens" principles of the frequency system in an appropriate way. We shall leave this issue as one of the important challenges to be addressed in the future development of formal systems for algorithmic analysis. Leo Guibas suggests that it might be possible to design a system in which the global, "report everything that ever happens" demons of the frequency system are replaced by a more local concept. One might be able to treat demons as objects in the programming language, rather like generalized counter variables, which the user of the system could explicitly create and manipulate. In such a system, presumably, recursive procedures could be handled by creating different demon instances for each level of the recursion.

In the next and final section, we shall discuss some other challenges that future systems should also address.

What Next?

The chromatic plumbing metaphor, the concept of a frequentistic assertion, and the other machinery of the frequency system seem to address rather successfully the problem of formalizing the dynamic phases of algorithmic analyses. Our ability in several examples to demonstrate by formal manipulation the extremely close coupling between the text of a program and the recurrence that determines its performance parameters is one of the frequency system's strongest selling points. But there are many directions in which further research should proceed.

As mentioned in the last section, it would be desirable to integrate recursive procedures cleanly into the frequency system framework. A good first step in this direction might be to extend Kozen's semantics to handle recursive procedures; presumably, the interpretation of a system of recursive procedures would turn out to be the least fixed point of a corresponding system of transformations.

It would be very desirable to have some completeness results at several levels. First, the assertion calculus should be specified more precisely, and some information gleaned about how close to complete it can be made. Actually, one would want to study the question of whether an assertion calculus was *relatively complete*, that is, complete if all true formulas in the underlying predicate calculus are considered as axioms. The relative completeness of an assertion calculus might turn out to be quite a subtle property, since, after all, a measure is assumed to be countably additive.

Then, at the next level, one would like to the show the relative completeness of rules of the frequency system itself, where "relative" here means that all true statements of the underlying assertion calculus are considered as axioms. This task might also prove tricky. We have noted that our version of the frequency system is incomplete, essentially because of the clumsiness of set operations.

Another important goal is a more powerful but still tractable assertion language. In particular, it would be good to be able to describe a random permutation in the discrete model. For such applications as the study of random hashing schemes, where the elements of a permutation are used as pointers into an array, the ability to handle a continuous model of random permutations does not seem to be any help; only a discrete model will do. If a single system could describe random permutations under both the discrete and continuous models, it might be possible to prove a metatheorem that demonstrated the equivalence of the two models. That is, one might be able to massage a derivation using one model according to certain rules, and turn it into a derivation using the other model.

There are new and perhaps good ideas emerging in the field of program verification today. Vaughan Pratt's *dynamic logic* [11, 12] and Manna and Waldinger's *intermittent assertions* [26] are two examples. It might be possible to build a system for average case algorithmic analysis based upon some of these post Floyd-Hoare ideas.

Finally, it would be interesting to consider in greater detail the question of formalizing subtle worst case arguments. The kinds of insights and techniques that we have been studying do not seem to be relevant, but perhaps some other approach would give a good formal handle on the reasoning behind analyses of worst cases.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] Boris Beizer. Analytical techniques for the statistical evaluation of program running time. In *Proc. AFIPS 1970 Fall Joint Computer Conf.* 37, Houston TX: 519–524. AFIPS Press, 1970.

[3] Jacques Cohen and Carl Zuckerman. Two languages for estimating program efficiency. *Comm. ACM* 17(6): 301–308, 1974.

[4] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Comm. ACM* 22(5): 271–280, 1979.

[5] Ole-Johan Dahl. Can Program Proving be Made Practical?. Lectures presented at the EEC-CREST course on Programming Foundations, Toulouse, 1977; revised May 1978. Oslo University Informatics Institute research report 033, 1978.

[6] N. G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symp. on Automatic Demonstration 1968*, Versailles: 29–61. Volume 125 of A. Dold and B. Eckmann, editors, *Lecture Notes in Mathematics*, Springer-Verlag, 1970.

[7] Edsger W. Dijkstra. Programming: from craft to scientific discipline. In E. Morlet and D. Ribbens, editors, *International Computing Symp. 1977*, Liege, Belgium: 23–30. North-Holland, 1977.

[8] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics* **19**, Providence, RI: 19–32. American Mathematical Society, 1967.

[9] Cordell Green. The design of the PSI program synthesis system. *Proc. 2nd International Conf. Software Engineering*, San Francisco, CA: 4–18, 1976.

[10] Paul R. Halmos. *Measure Theory.* Van Nostrand, 1950.

[11] David Harel. *Logics of Programs: Axiomatics and Descriptive Power.* PhD thesis, Massachusetts Institute of Technology, 1978. Also published as report MIT/LCS/TR–200.

[12] D. Harel, A. R. Meyer, and V. R. Pratt. Computability and completeness in logics of programs (preliminary report). In *Proc. 9th ACM Symp. Theory of Computing*, Boulder CO: 261–268, 1977.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM* **12**(10): 576–580 and 583, 1969.

[14] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica* **2**: 335-355, 1973.

[15] Dan Ingalls. The execution time profile as a programming tool. In Randall Rustin, editor, *Design and Optimization of Compilers*, Courant Computer Science Symp. 5, 1971: 107–128. Prentice-Hall, 1972.

[16] Arne T. Jonassen and Donald E. Knuth. A trivial algorithm whose analysis isn't. *J. Computer and System Sciences* **16**(3): 301–322, 1978.

[17] Elaine Kant. *Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach.* PhD thesis, Stanford University, 1979.

[18] Donald E. Knuth. *Fundamental Algorithms*, Sections 1.2.1 and 1.2.10. Volume 1 of *The Art of Computer Programming.* Addison-Wesley, second edition 1973.

[19] Donald E. Knuth. Mathematical analysis of algorithms. In volume 1 of *Proc. of 1971 IFIP Congress*, Ljubljana, Yugoslavia: 19–27. North-Holland, 1972.

[20] Donald E. Knuth. *Sorting and Searching*, Section 5.2.1. Volume 3 of *The Art of Computer Programming.* Addison-Wesley, 1973.

[21] Donald E. Knuth. *Tau Epsilon Chi: A System for Technical Text.* American Mathematical Society, 1979. An earlier version appeared as Stanford University report STAN–CS–78–675, 1978.

[22] Dexter Kozen. Semantics of probabilistic programs. IBM Thomas J. Watson Research Center report, Computer Science, RC 7581 (#32819), 1979.

[23] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof Rules for the Programming Language EUCLID. *Acta Informatica* **10**(1): 1–26, 1978.

[24] David C. Luckham and Norihisa Suzuki. Proof of termination within a weak logic of programs. *Acta Informatica* **8**(1): 21–36, 1977.

[25] Zohar Manna. *Mathematical Theory of Computation.* Mc-Graw Hill, 1974.

[26] Zohar Manna and Richard Waldinger. Is "sometime" sometimes better than "always"? Intermittent assertions in proving program correctness. *Comm. ACM* **21**(2): 159–172, 1978.

[27] Zohar Manna and Richard Waldinger. The logic of computer programming. *IEEE Trans. Software Engineering* SE–4(3): 199–229, 1978.

[28] The Mathlab Group, Laboratory for Computer Science, MIT. *MACSYMA Reference Manual.* Massachusetts Institute of Technology, version nine, second printing, 1977.

[29] Elliott Mendelson. *Introduction to Mathematical Logic.* Van Nostrand, 1964.

[30] C. V. Ramamoorthy. Discrete Markov analysis of computer programs. In *ACM 20th National Conf.*, Cleveland OH: 386–392, 1965.

[31] E. Satterthwaite. Debugging tools for high level languages. *Software—Practice and Experience* **2**: 197–217, 1972.

[32] L. S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH System.* PhD thesis, Technological University Eindhoven, The Netherlands, 1977.

[33] Ben Wegbreit. Verifying program performance. *J. ACM* **23**(4): 691–699, 1976.

[34] Ben Wegbreit. Mechanical program analysis. *Comm. ACM* **18**(9): 528–539, 1975.

XEROX